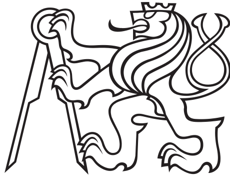
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Managment System for Georeferenced Unmanned Aerial Vehicle Sensory Data

Milan Zelenka

January 2017
Supervisor: Ing. Milan Rollo Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Milan Zelenka

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Systém pro správu georeferencovaných senzorických dat z bezpilotních prostředků

Pokyny pro vypracování:

Bezpilotní prostředky nalézají stále vyšší uplatnění při sběru senzorických dat v různých oblastech. Většina senzorických dat je vztažena ke konkrétní oblasti a je možné k nim dále přiřadit i řadu metainformací z telemetrických údajů. Velké množství a objem dat způsobuje problémy při jejich ukládání, katalogizaci a následném zpracování.

1) Prozkoumejte problematiku georeferencování senzorických dat získaných pomocí bezpilotních prostředků.
2) Zpracujte přehled existujících řešení, a to jak komerčních tak open source.
3) Navrhněte a implementujte systém pro georeferencování dat z palubních kamer a jejich ukládání.
4) Implementujte algoritmy pro vyhledávání objektů zájmu (ve fotkách a videích) daných jejich polohou (GPS souřadnice, nadmořská výška).
5) Vytvořený systém otestujte na reálných datech.

Seznam odborné literatury:

[1] Hemerly E. M.: Automatic Georeferencing of Images Acquired by UAV's. International Journal of Automation and Computing, 11 (4), pp. 347-352, 2014.
[2] J.-G. Lee, M. Kan: Geospatial Big Data: Challenges and Opportunities. Big data research, vol. 2, issue 4, pp. 74?81, 2015.

Vedoucí: Ing. Milan Rollo, Ph.D.

Platnost zadání do konce zimního semestru 2017/2018

L.S.

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 12. 9. 2016

/ Declaration

I do hereby declare, that I developed the submitted thesis independently, and I stated every piece of information sources which I used according to the methodical instructions about compliance of ethical principles during the development of a master's thesis.

Abstrakt / Abstract

Tato práce se zabývá georeferoncováním sensorických dat získaných z bezpilotních vzdušných prostředků. Navrhuje řešení, jak tato data efektivně ukládat a jak vyhledávat takové části těchto dat, které obsahují objekty zájmu definované svou geografickou pozicí. Demonstruje, jakým způsobem byla data sbírána spolu s problémy, jenž vyvstávaly v důsledku jejich nepřesností, ukládání těchto dat v databázi, mapování prostoru s jejich pomocí a popisuje způsob, jak vzhledávat ta data, která obsahují dané objekty zájmu.

Klíčová slova: bezpilotní letouny, georeferencování, geo-prostorová databáze, PostgreSQL, PostGIS, mapování geografických dat, GoPro kamera, kalibrace kamery, odometrie

Překlad titulu: Systém pro správu georeferencovaných senzorických dat z bezpilotních prostředků

This work deals with georeferencing of sensory data acquired by unmanned aerial vehicles. It proposes a solution of how to store these data effectively and how to seek out parts of these data containing objects of interest given by their geographical position. It deals with data acquisition and the setbacks which arose due to their inaccuracies, storing of these data in a database, using them to map the real environment and describes approaches of how to seek out their parts containing objects of interest.

Keywords: UAS, UAV, drones, georeferencing, geo-spatial database, PostgreSQL, PostGIS, geo-mapping, GoPro, camera calibration, odometry

Contents

1 Introduction ... 1
1.1 Applications of UAS ... 1
1.2 Motivation and goals ... 2
1.3 Thesis structure ... 2
2 Current State of the Art ... 3
2.1 LiDAR sensors ... 3
2.2 Photogrammetry ... 4
2.3 Videos and images ... 4
2.4 Data storage ... 5
2.5 Spatial databases ... 6
2.5.1 PostGIS performance ... 7
2.5.2 Comparison with a commercial solution ... 8
3 Design ... 9
3.1 Data acquisition - components ... 10
3.1.1 u-blox NEO-M8N ... 10
3.1.2 ArduPilot Mega 2.6 ... 11
3.1.3 GoPro HERO 4 Black ... 12
3.1.4 Data acquisition - format ... 12
3.2 Data mapping ... 13
3.2.1 Mapping algorithm ... 13
3.2.2 Additional filtering ... 14
3.3 Data storing ... 14
4 Implementation ... 16
4.1 Telemetry acquisition ... 17
4.2 Camera calibration ... 17
4.2.1 Calibration procedure ... 18
4.2.2 Calibration results ... 20
4.3 Odometric calculations ... 21
4.3.1 Feature detection ... 21
4.3.2 Feature matching ... 22
4.3.3 Essential matrix ... 23
4.4 Data mapping ... 24
4.5 Mapping algorithm ... 25
4.5.1 Bounding polygon ... 25
4.5.2 Limiting the distance of view ... 29
4.5.3 Translation of the bounding polygon ... 29
4.5.4 Additional filtering ... 30
4.5.5 3-dimensional check ... 31
4.6 Database structure ... 32
4.6.1 Additional adjustments ... 34
5 Experiments ... 35
5.1 Visual experiment ... 35
5.1.1 Visual experiment - results ... 36
5.2 Field of view experiment ... 37
5.2.1 Field of view experiment - results ... 38
5.3 Sensor correctness experiment ... 39
5.3.1 Sensor correctness experiment - results ... 39
5.4 Visual experiment revisited ... 39
5.4.1 Visual experiment revisited - results ... 40
5.5 Scalability ... 41
5.5.1 Search ... 41
5.5.2 Insert ... 42
6 Conclusion and future work ... 44
References ... 46

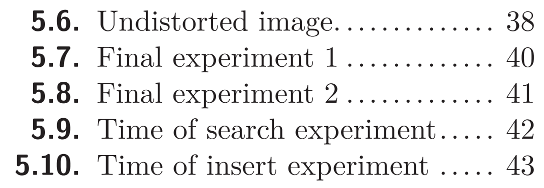
Tables / Figures

Tables

2.1. MySQL vs PostGIS - functionalities 7
3.1. *NEO-M8* specifications 11
3.2. *GoPro FOV* specifications 12
4.1. *Calibration matrix* 20
4.2. *FOV of rectified a image.* 21
4.3. *Elevation error* 25
5.1. *Drone parameters* 36
5.2. FOV measurements 39
5.3. Time of search experiment 42
5.4. Time of insert experiment 43

Figures

1.1. Drone spraying pesticides 1
2.1. Point cloud 4
2.2. Photogrammetry example 4
2.3. Filesystem and BLOBs - fragmentation effects 5
2.4. Filesystem and BLOBs - read throughput 6
2.5. MySQL vs PostGIS - contains 6
2.6. Bounding box visualization 7
2.7. R-tree visualization 8
3.1. Work flow diagram 9
3.2. Y6 hexacopter 10
3.3. Y6 hexacopter wiring 10
3.4. u-blox NEO-M8N 11
3.5. ArduPilot Mega 12
3.6. GoPro HERO 4 Black 12
3.7. False detection 14
3.8. Simplified database schema 15
4.1. Point of interest search flow 16
4.2. Barrel distortion 17
4.3. Calibration - corner extraction 18
4.4. Distortion estimate - before 19
4.5. Distortion estimate - after 19
4.6. Reprojection error 20
4.7. Distorted image. 21
4.8. Undistorted image. 21
4.9. Feature detection 22
4.10. Feature matching 23
4.11. Coordinate system transformation 24
4.12. Curvature of Earth error 24
4.13. Rays facing down 26
4.14. Rays rotated 27
4.16. Convex hull - example 28
4.17. Visible area - frustum 28
4.18. Visible area - bounding polygon 29
4.19. Monitored area - convex hull 31
4.20. 3-dimensional check 32
4.21. Database E-R diagram 32
5.1. Comparing data - visualizer 35
5.2. Compering data - video 35
5.3. Altitude error - viewed area 37
5.4. Field of view calculation 38
5.5. Distorted image. 38



5.6. Undistorted image. 38
5.7. Final experiment 1 40
5.8. Final experiment 2 41
5.9. Time of search experiment 42
5.10. Time of insert experiment 43

Chapter 1
Introduction

Unmanned Aerial Systems (UAS), in this work's context also referred to as drones, have become an attractive data acquisition platform in emerging applications. As measuring instrument they extend the lineup of possible surveying methods in the field of geomatics [1]. With the UAV hardware decreasing in its size and becoming more affordable, it is now possible to mount these vehicles with advanced payload and use them for computationally extensive tasks.

UAVs may operate at various levels of autonomy: either under remote control by a human operator, or fully or intermittently autonomously, controlled by onboard computers [2].

1.1 Applications of UAS

When used, UAS often perform missions characterized by the *three Ds*: dull, dirty, and dangerous. Dull means long-endurance missions which, in the future, could continue for several days. Dirty means jobs such as detecting chemical agents and their intensity, which, due to different dangers, should not be interacted with directly. Dangerous missions for unmanned vehicles are numerous and growing and can be usually found in the context of warfare [3]. Civilian applications of use may extend from agriculture - e.g. figure 1.1, delivery of goods, journalism, safety inspections to police surveillance; civilian drones now vastly outnumber the military ones.

Figure 1.1. A drone sprays pesticides on a farm in Bozhou, central China's Anhui province [4].

1.2 Motivation and goals

The motivation of this work is to design and implement a process of sensory data georeferencing and store these data in a persistent storage in such a way that allows for effective search based on geographical coordinates - longitude, latitude and altitude.

Typical scenario would be gathering of sensoric data in form of photos or videos (e.g. for the purposes of surveillance) along with the camera's geographical coordinates as well as it's orientation in space (azimuth, roll, pitch). An operator would be controlling a UAS with a mounted camera possibly with a gimbal and he or she would be monitoring objects of interest in an area. Based on these data, the viewed area of these images or videos would be calculated and stored in a persistent storage. This storage should then allow for a search of arbitrary objects (points of interest) contained in these photos or videos based on their geographical position. The system should then be able to seek out all photos and videos containing such a point of interest in the entire database.

1.3 Thesis structure

The thesis is broken down into several parts and is organized as follows.

The chapter called *Current State of the Art* talks about the current technologies used for similar purposes to the ones of this work and the means which can be utilized to achieve this work's goals.

The subsequent chapter titled *Design* informs the reader about the technologies, concepts and the main ideas used to implement the final product. However, it does not go into detail and gives the reader solely a brief overview of what is described in detail in the next chapters.

The following chapter named *Implementation* describes in detail the approaches and methods used in this work. It is organized in the following manner.

- At the beginning it describes the methods by which the raw data is acquired, its format and the way it is processed. It talks about inaccuracies of data measurements and possible ways of working around them.

- Afterwards, it shows a solution of how to map the acquired data to images and videos and describes the calculations and algorithms used.

- Finally, it proposes a way of effective data storing with focus on search. It explains why such a way is effective and the reasons why it was used.

The final chapter then describes the experiments used throughout this work along with their results and provides reasoning why it rendered to be successful or unsuccessful.

Chapter 2
Current State of the Art

In the recent years, due to the rapid development of sensors and information technologies in general, a lot of research has been done in the georeferencing field. Georeferencing itself means assigning geographical coordinates to some data such as in this work's context to sensory data. This process may be done in different ways and is reflected in its data acquisition. Some of the ways of how to acquire the data to be georeferenced is described in this chapter.

Also, these data need to be stored in a way which allows us to work effectively with it. This chapter explores some of the currently used mechanisms allowing for such a data storage.

2.1 LiDAR sensors

One of the considered acronyms of LiDAR is *Light Detection And Ranging*. One can picture a LiDAR as a device for distance measuring using a laser beam. The wavelength used varies depending on the type of objects measured. Below are listed some examples of use.

- infrared (1500 - 2000 nm) for meteorology
- near-infrared (1040 - 1060 nm) for terrestrial mapping
- blue-green (500 - 600 nm) for bathymetry
- ultraviolet (250 nm) for meteorology

Some of the available LiDARs can measure distances up to 1.5 kilometers and can target many different types of materials [5]. Consequently, and due to accessibility of drones, airborne LiDARs have become rather popular. These are able to create three-dimensional models of environment called *point clouds* that represent the scanned landscape 2.1.

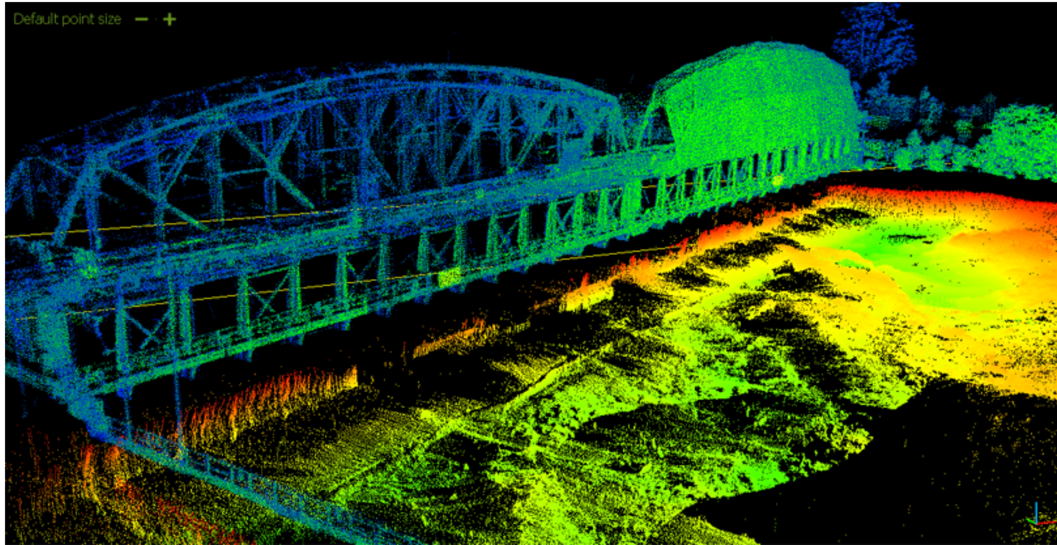


Figure 2.1. A point cloud created by a LiDAR [6].

2.2 Photogrammetry

Photogrammetry is an approach of how to map a scanned environment and can be described by a human eye perception analogy. The photogrammetric system tries to estimate the relationships between image and object space based on visual data using disciplines such as optics and projective geometry. An example of a model acquired using photogrammetry is depicted by figure 2.2. An advantage is that most common cameras can be used for photogrammetry, however, the necessary processing time is significantly higher than it is for LiDAR scanning and the data acquisition is also often relatively time consuming as enough images of the object of interest have to be taken in order for an accurate model to be created.

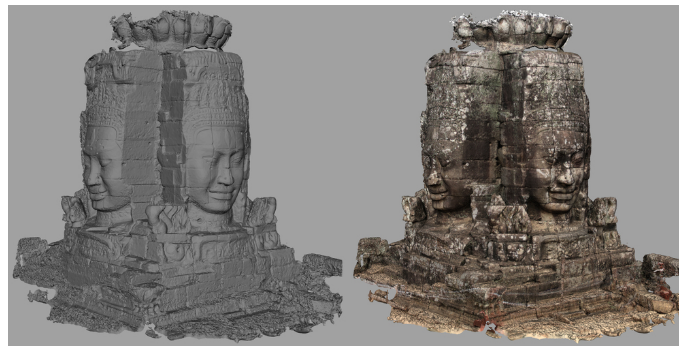


Figure 2.2. A model (left) created by photogrammetric methods [7].

2.3 Videos and images

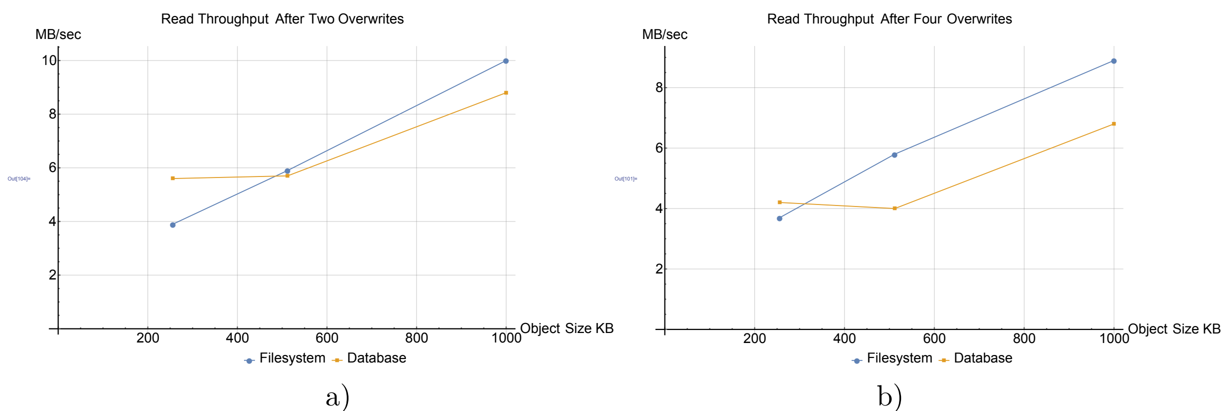
Another way of how to store sensory data is in form of raw images and videos. The advantage is they do not require any technology other than a camera but they usually do not carry other relevant information. Therefore, for the purposes of georeferencing, it is necessary to store other necessary information such as the camera's GPS coordinates separately and implement a way of mapping them to the video.

Since the main goal of this work is to design a system which is able to seek out objects mainly in videos and pictures based on their GPS position, this approach was chosen as the optimal one with little overhead to its processing time.

2.4 Data storage

The data that is being dealt with in this work, as stated previously, are videos and images as well as information about the UAS bearing a camera such as its GPS coordinates and orientation as the videos or images are taken. Since the UAS' information can be stored as plain text and does not require too much space on the disk, it can be stored in a relational database. The question arises whether the images and videos should be stored in a database as BLOBs (binary large obejcts) or in a filesystem.

When deciding whether to store files in a filesystem or a database, storage fragmentation is the main determinant and filesystems seem to have better fragmentation handling than databases. However, this advantage only applies for files that are larger than 256kb and if the objects are under 256kb, the database has a clear advantage [8]. Figure 2.3 approximately depicts when a database outperforms a filesystem and vice versa. This experiment was done using *Microsoft SQL Server* and the *NTFS* filesystem.



a) b)

Figure 2.3. Fragmentation effects on different types of storage [8].

When we consider the idea that no files are overwritten, which might be a possible scenario in our system's context, the filesystem starts outperforming the database once the files are larger than 1MB as shown in figure 2.4. As this work relies on images and videos in particular that are generally larger than 1MB, it was decided to store these data in a filesystem [8].

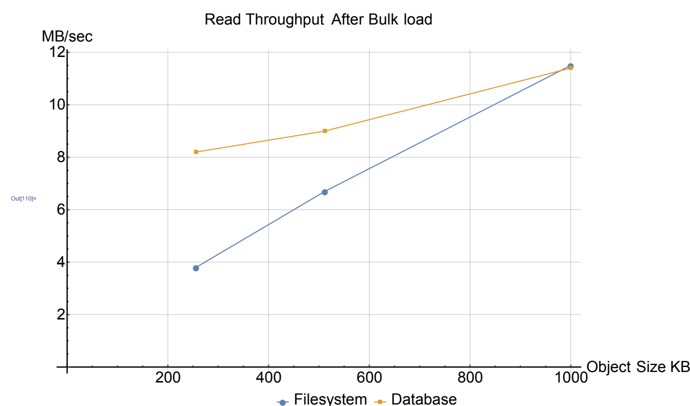


Figure 2.4. Read throughput immediately after bulk loading the data [8].

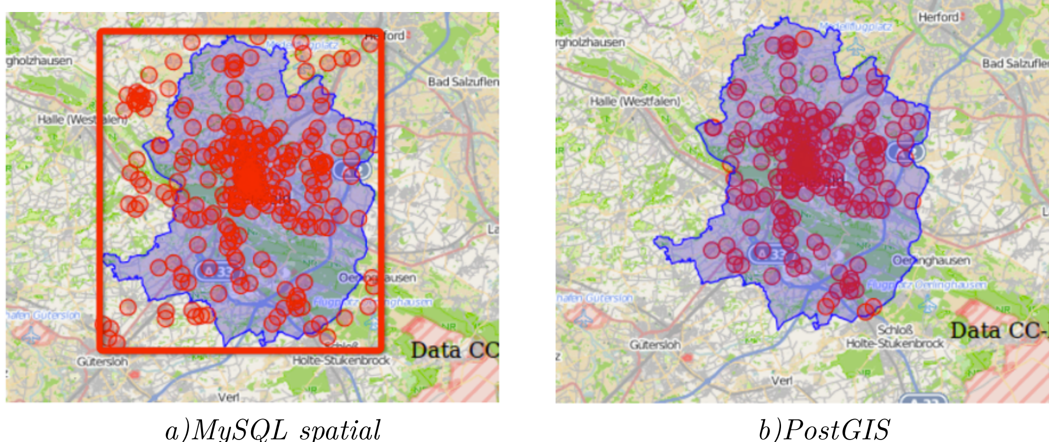
2.5 Spatial databases

A spatial database is a data store optimized for geometrical objects. Thereby, we are able to store our geographical data efficiently and work effectively with them due to spatial data indexing. Moreover, such databases often provide other useful functionalities with these data such as spatial joins, which allow us to append fields from a table to another table, or functions allowing us to perform different geometrical operations effectively such as checking whether one objects lies within another object [9].

Several spatial databases were considered to be used, however, all that were not free and hence not affordable were disqualified. Also, as it is necessary to store other than geographical data related to particular areas, a relational database was assumed to be the most convenient means of data storage.

SpatialLite is a spatial extension to SQLite, which is a lightweight file-based database. Its installation and setup are very simple and for the purposes of development it would prove to be sufficient, however, in respect to possible further scalability, such as possible concurrent connections or different database users, this option was discarded.

MySQL and its spatial extension enable the generation, storage and analysis of geographic features. It has a relatively extensive API and was one of the two main candidates to be used in this work. Compared to *PostgreSQL*, however, it was discarded as some of *PostgreSQL*'s features outperform the ones of *MySQL* as demonstrated by table 2.1. Also, some of *MySQL's* functions bear limitations such as querying whether a point lies inside a polygon; *MySQL* only finds whether a point lies inside a polygon's minimum bounding rectangle while PostGIS can detect any point inside this polygon as depicted in figure 2.5.



a)MySQL spatial *b)PostGIS*

Figure 2.5. *MySQL spatial* vs *PostGIS* contains functions [10].

Due to previously mentioned reasons, the database used in this work was *PostgreSQL*, which is an open source object-relational database with a spatial extension named *PostGIS*. It runs on all major operating systems and bears a reputation of being reliable. At the same time it has an extensive documentation with many examples of how it can be used and can be natively interfaced from most major programming languages [11].

	MySQL spatial	PostGIS
Spatial index	R tree	GIST tree (R tree varieties)
Space type	Only 2-D	2-D, 3-D and curve
Space projection	Does not support	Supports a variety of common projective coordinate systems

Table 2.1. MySQL vs PostGIS [10].

2.5.1 PostGIS performance

One of the major reasons why spatial databases are efficient with spatial data is that they provide means of geometrical and geographical data indexing. In *PostGIS* instead of exploring the attributes and relationships between particular data entries, we use their corresponding bounding boxes depicted by figure 2.6. Part *a* of the image shows different geometrical objects while part *b* depicts them along with their bounding boxes. Part *c* then shows the resulting objects that are indexed in the database. Not only do they occupy much less space in memory then their original shapes, but also mathematical operations are generally more simple with them. That way we can easily discriminate between the majority of data in our database and perform more accurate operations with them thereafter.

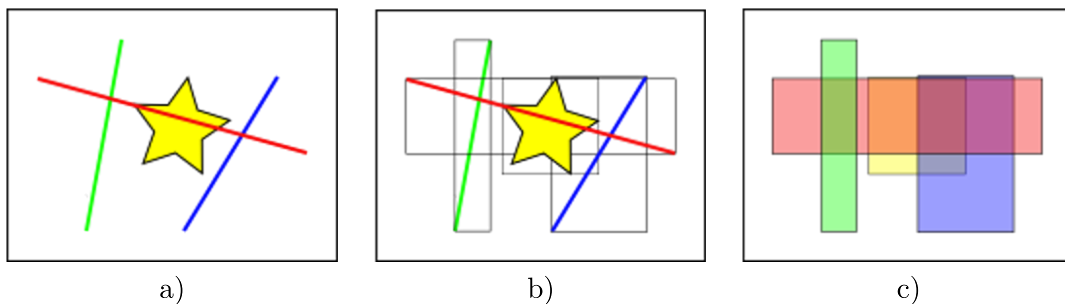


Figure 2.6. *PostGIS* bounding boxes visualization [12].

As a result, it is possible to store these bounding boxes in an *R-Tree* index visualized by figure 2.7; it is a height-balanced data structure similar to a *B-tree* used for multi-dimensional spaces [12]. Due to this, the average time complexity of a search is $O(log_M n)$ where M is the maximum number of entries in a node and n is the total number of entries. The time of each insert may vary and its time complexity is $O(n)$ in the worst case [13].

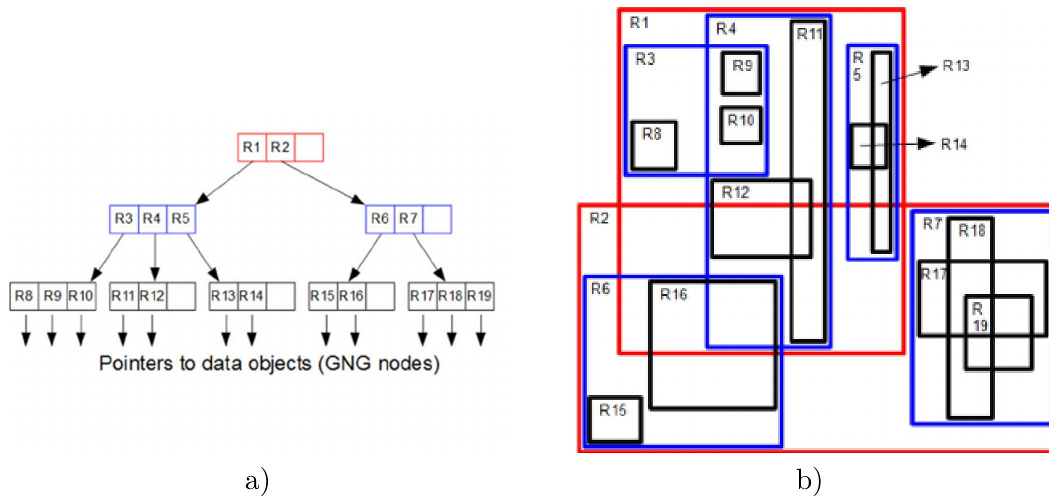


Figure 2.7. *R-Tree* visualization [14].

PostGIS provides us with many functions such as one which returns true if geometry B fully lies in geometry A; in a similar way there are other functions to be found, e.g. querying whether two geometries intersect, if they form a ring or one that constructs a convex hull of a set of points and many more.

2.5.2 Comparison with a commercial solution

A commertial alternative to *PostGIS* is *Oracle Spatial*. One might expect an IT company of such a size to perform better than an open source development community of a few dozen core developers, but due to results of a benchmark [15] comparing Oracle 11g and Postgres 9.04/Postgis 1.5.2 that is not the case. The authors of this benchmark claim:

From the experimental results that we saw, we can conclude that Postgres performs better than Oracle 11g both in the Cold Phase and Warm Phase. Though in few queries Oracle 11g performed better but on the whole Postgres overpowered Oracle 11g. In the warm phase in 3 out of 4 queries Postgres performed significantly well, from this we can conclude that Postgres has better automatic memory management capabilities and page replacement policies. May be Oracle 11g needs more tuning to perform better. And also in the Cold Phase, Postgres performs significantly well except in few cases such as in the Adjacency Operation (TOUCH). Since Postgres uses the underlying GEOS(Geometry Engine - Open Source) library functions for implementing the geometric operations whereas Oracle 11g implements them on its own, and since in majority Postgres performs well, we can conclude that GEOS geometric algorithms are more efficiently designed than Oracle 11g. And also Postgres planner is more efficiently designed to take advantage of any available indexes to use in queries for achieving better performance whereas in Oracle 11g we saw that we have to specify them explicitly through functions. On the whole it is the open-source that wins the game [15]!

There are other factors affecting the outcome such as that *PostGIS* was run on *Linux* while the Oracle database was run on *Windows* or neither did the author try to optimize the databases' environments; therefore both were run with default parameters, which might otherwise change the outcome entirely.

Chapter 3
Design

This chapter describes the general work flow of the proposed solution and will try to familiarize the reader with all its relative concepts; chapter 4 will then describe these processes in detail.

The goal of this work is to allow for repeatable area monitoring and surveillance by an operator using a UAS with a mounted camera. This operator would guide a UAS through an area and capture images or videos, which would then be stored in a persistent storage along with other relevant data. The area may be scanned by multiple cameras and at different times such as on weekly/daily basis and there might be multiple areas that are monitored - later referred to as *Monitored areas*. However, the system is not limited only to UAS controlled by an operator, but to any sensory data that comes along with all the necessary information required for the algorithms described in this work, such as stationary cameras or automated UAS.

Afterwards, it should be possible to seek out arbitrary objects in these videos and images based on their geographical coordinates across the entire database with the possibility of additional filters, such as the time when the data were taken, the monitored area the data were taken in or the direction the object of interest was viewed from.

As the main purpose of this work is to enable an arbitrary object look up in sensory data, these data also need to describe the area being captured. This process can be divided into several phases and all of them are described in this chapter; the diagram 3.1 serves as its outline.

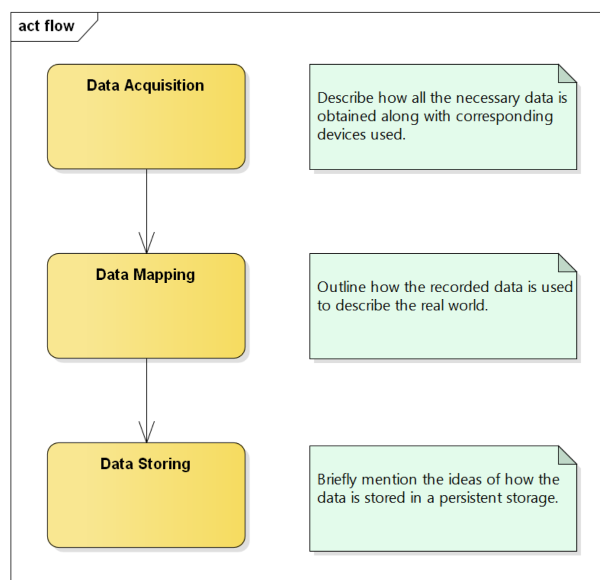


Figure 3.1. Work flow diagram.

3.1 Data acquisition - components

All data acquisition was done using a drone (Y6 hexacopter) depicted by figure 3.2, composed of multiple components described below. Its complete schema is depicted in figure 3.3.

a) b)

Figure 3.2. Y6 hexacopter used to acquire data.

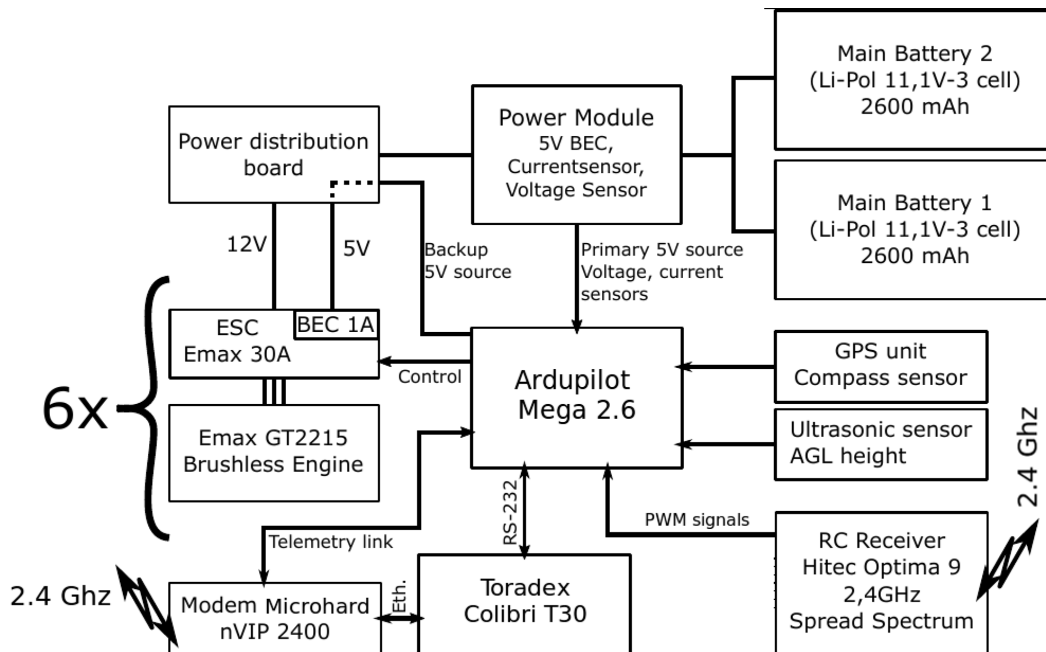


Figure 3.3. Wiring of the Y6 hexacopter [16].

3.1.1 u-blox NEO-M8N

The *NEO-M8* series utilizes concurrent reception of up to three GNSS systems - GPS/Galileo, BeiDou and GLONASS and recognizes multiple constellations simultaneously to provide better postitioning accuracy in scenarios where urban canyon or weak signals are involved. The NEO-M8N depicted in figure 3.4 offers high performance also at low power consumption leves, which makes it a suitable candidate for drones where the power supply is limited [17].

Parameter	Specification	Value
Horizontal position accuracy	Without aiding	2.5 m
Horizontal position accuracy	SBAS	2.0 m
Velocity accuracy		0.05/s
Heading accuracy		0.3 degrees

Table 3.1. *NEO-M8* specifications [17].

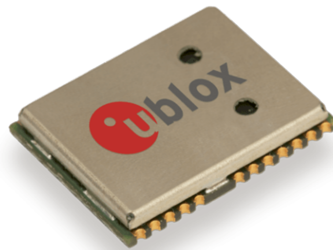


Figure 3.4. NEO-M8 series - 12.2 x 16 x 2.4 mm [18].

3.1.2 ArduPilot Mega 2.6

Ardupilot Mega (APM) depicted in figure 3.5 is a professional quality IMU autopilot that is based on the Arduino Mega platform. This autopilot can control fixed-wing aircraft, multi-rotor helicopters, as well as traditional helicopters. It is a full autopilot capable for autonomous stabilisation, way-point based navigation and two way telemetry with Xbee wireless modules [19]. It has an open source autopilot firmware supporting planes, multicopters, helicopters and ground rovers with a logging memory providing 4MB of space.

All the rotations that are necessary for the camera's calculations are obtained from the autopilot's Inertial Measurement Unit (IMU) as for the purposes of development the camera was fixed underneath the drone in a constant angle; that way all the drone's rotations can be translated to the camera. It should be noted the camera is positioned approximately 20 cm away from the IMU which is why during the pitch and roll rotations[1] its position is slightly off-set from NEO-M8N. However, during development this off-set was neglected as the positioning accuracy range of NEO-M8N is far bigger then this off-set. If there was a need to use a gimbal for the camera in the future, its rotations need to be taken into account.

[1] These rotations are explained in section 4.5.

Figure 3.5. ArduPilot Mega [19].

3.1.3 GoPro HERO 4 Black

GoPro HERO 4 Black depicted in figure 3.6 is a small action camera offering high quality imaging along with variable parameters, that can be set by their users, such as field of view (FOV), resolution, frame rate, etc. It is both light and small, thereby it can be conveniently mounted on drone without significantly changing its flight properties. Despite this, it also offers sufficient battery life and disk space. Using wide field of view, however, applies rather a significant distortion to its images that needs to be taken into account and in some cases corrected.

	V.FOV deg	H.FOV deg	Diag. FOV deg
4 x 3 Wide	94.4	122.6	149.2
4 x 3 Medium	72.2	94.4	115.7
4 x 3 Narrow	49.1	64.6	79.7
16 x 9 Wide	69.5	118.2	133.6
16 x 9 Medium	55	94.4	107.1
16 x 9 Narrow	37.2	64.4	73.6

Table 3.2. *GoPro* Field of View [20].

The sensor dimensions are 6.17 mm x 4.55 mm, which might be useful while calculating the camera's focal length as GoPro's specifications available online [20] list only focal length equivalent for all FOVs (e.g. 9.5-11mm range), which are 17.2mm, 21.9mm and 34.4mm for wide, medium and narrow FOV respectively. Further on, it will be explained how the real focal length was calculated after camera calibration had been performed.

Figure 3.6. GoPro HERO 4 Black [21].

The GoPro camera also offers a wide variety of image resolutions (up to 4k) along with adjustable frame rate ranging from 24 up to 240 fps; not all the resolutions permit all fps settings, however.

3.1.4 Data acquisition - format

The flight data is recorded in a file by the autopilot but for purposes of this work and for the reason that the output file includes other unnecessary entries, the data needs to be filtered out. Also, as it is convenient to assign the camera's current attitude

to individual video frames and as it is not always received at the time of each frame, particularly when dealing with high frame rates, it is required to interpolate the data between two attitude entries.

The resulting data set then consists of entries each having a timestamp, longitude, latitude, altitude, roll, pitch and yaw.

3.2 Data mapping

Since this work uses the *PostGIS* database to store data, it was necessary to decide, which coordinate system to use. *PostGIS* offers two spatial types called *geography* and *geometry*. *Geography* uses geodetic measurements contrary to *geometry*, which uses Cartesian measurements. The advantage of *geometry* is that it provides a richer set of functions than *geography* and is generally faster since calculations on a sphere are more complicated. For these reasons *geometry* was selected as the used format.

The trade-off of this approach are imprecisions that arise due to coordinate conversions as explained in section 4.4. It is up to further studies, whether the performance and functionality improvements of *geometry* can be out-weighted by using the more precise *geography* format.

Consequently, the recorded data have to be categorized into so called *Monitored Areas* as explained at the beginning of this chapter. Each of these areas have their own Cartesian coordinate system with its origin specified in an arbitrary GPS coordinate in this area, which is then necessary for any other conversion from GPS to Cartesian coordinates in this area. In this work, the first coordinate ever recorded in a given area is used as its orgin.

3.2.1 Mapping algorithm

Basic algebraic approaches were used to describe the area being captured and are further explained in chapter 4. In principle, 4 rays are created outlining the corners of the camera's field of view. Then we look for intersections of these rays with the ground; if the camera's rays look over the horizon and thus they intersect the ground behind the camera, a limiting distance is set defining the farthest viewed area. This approach is explained more in detail in the following chapter in section 4.5.2. Afterwards, a convex hull of these points and the camera's current position is built, which defines a polygon representing the viewed area projected on the ground. This polygon is then saved to the database along with the camera's attitude for each image or each video frame of a video that is stored.

The reason for this simplification is that it can be discriminated between frames and images definitely not containing a point of interest and frames possibly containing it doing only 2-dimensional operations. These 2-dimensional operations are fast, particularly due to *PostGIS'* spatial index and functions that can find 2-dimensional objects containing a point, compared to more complicated 3-dimensional operations, especially while working with large data sets. Afterwards, using the smaller processed dataset, a more precise check is performed whether the point of interest lies within the camera's field of view by forming 4 planes surrounding it and computing whether the point of interest lies within it. Figure 3.7 depicts a scenario when two objects are possibly lying

within the camera's field of view, however, only one (the hexagon) will pass through a 3-dimensional check.

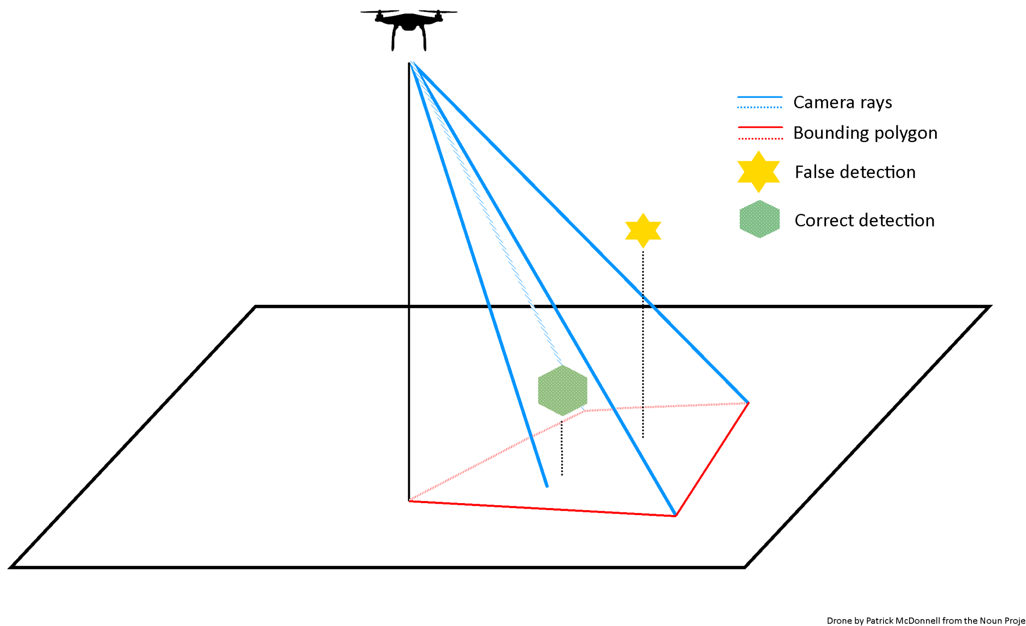


Figure 3.7. False 2d detection.

3.2.2 Additional filtering

As one database may contain data gathered from multiple *Monitored areas* that might otherwise be unrelated and in order to be able to discriminate between these areas and to minimize the data set a point of interest is being searched in, a database entity representing a monitored area was created. This entity has an attribute *bounding_polygon* representing the maximum area covered by all video frames or images bound to this area described in the GPS coordinates. As new frames are added to the database, it is necessary to adjust the *bounding_polygon* accordingly. This slows down the post-processing time when adding new data to the database, however, the search time is improved. The advantage of this approach is that we can either query the database for all areas possibly containing a point of interest and it is possible to verify whether a monitored area contains one.

3.3 Data storing

As stated earlier the *PostgreSQL* database along with the *PostGIS* extension was used to store data; a simplified model shown in figure 3.8 describes the idea of how it is done. As areas, in this work's context, are meant to be monitored over time, an entity *MonitoredArea* was created that groups all *DataSets* related to them. Each *DataSet* then groups all its corresponding video frames or images. Using the *IS-A* database approach [22], it is possible to store both picture sets and videos.

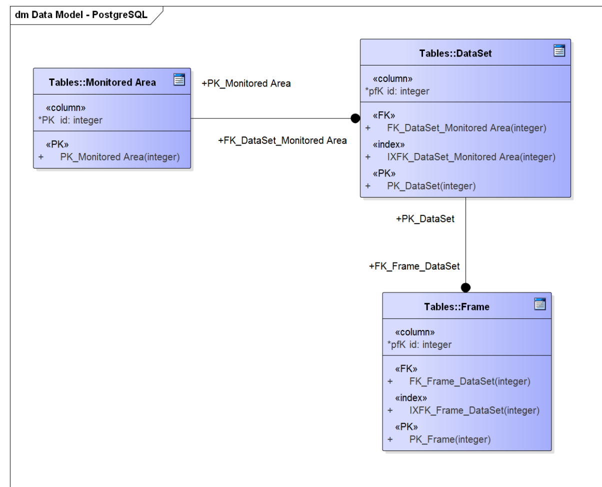


Figure 3.8. Simplified database schema.

Chapter 4
Implementation

In this chapter, the algorithm used to find a point of interest in the gathered sensory data will be described in detail along with technical details accompanying the development. The algorithm seeking out data with a point of interest is demonstrated by diagram 4.1.

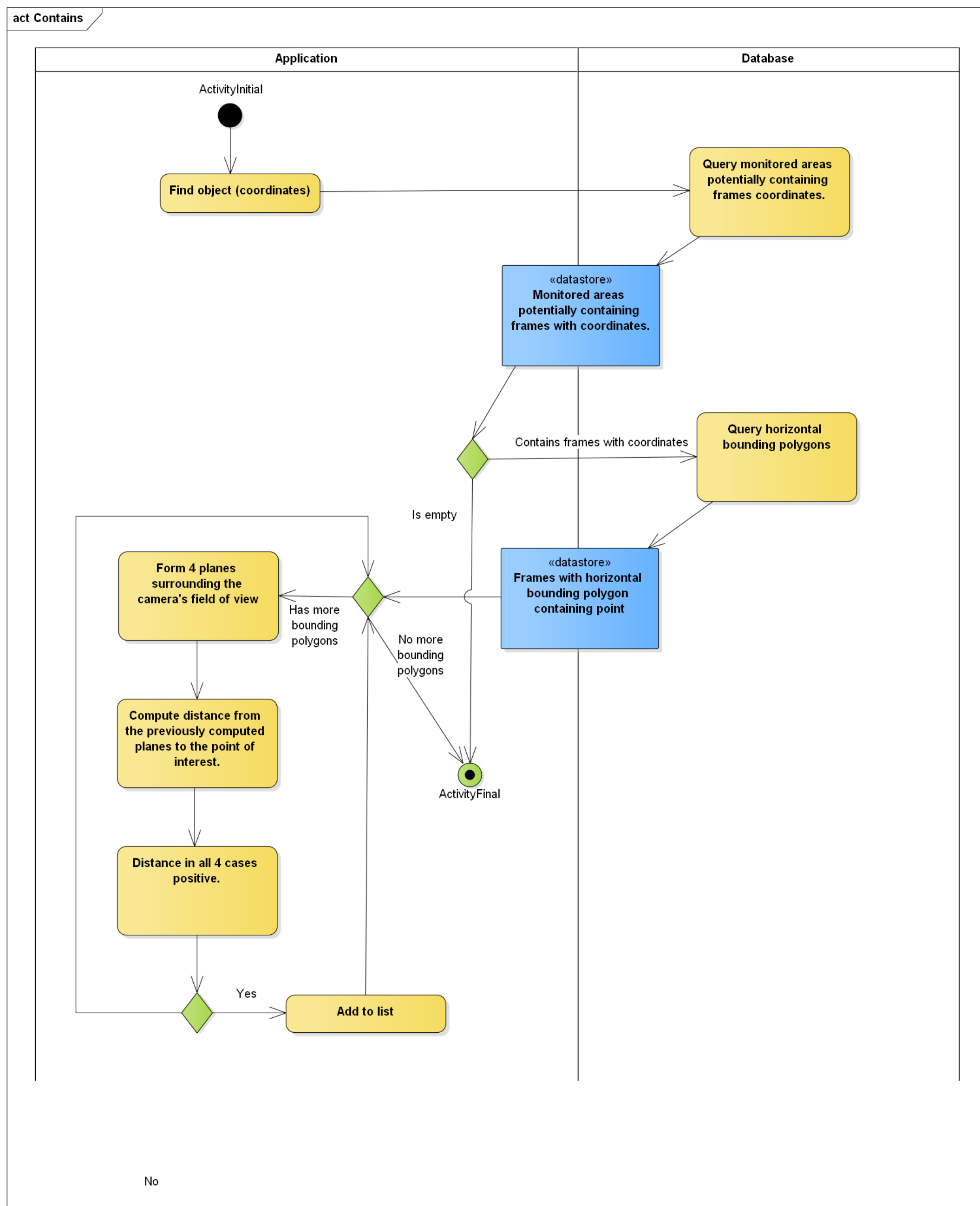


Figure 4.1. Point of interest search flow.

This chapter explains the proposed solutions while the following chapter *Experiments* deals with their practical implementation and demonstrates their results. First, it will describe how the necessary data were acquired, afterwards how they were processed and in the end the approach of how they were stored.

4.1 Telemetry acquisition

The telemetry is collected by components described in chapter 3. The resulting data file consists of entries recorded at different times. Each of them consists of different information, such as rotations in axes, GPS coordinates with time and the current uptime. As we need the real world time of each entry, using a *MATLAB* script we parse these data and using the GPS time contained in some entries and the autopilot's uptime contained in all entries, we interpolate the real time of all of them.

4.2 Camera calibration

Another important data component are images and videos. Throughout development it became clear that additional operations with images and videos would be necessary, therefore there was a need to find out the camera's intrinsic parameters.

In order to get the exact parameters of the *GoPro* camera such as its horizontal and vertical field of view along with its principal point of view and its distortion coefficients, it was necessary to perform a camera calibration. Also there was a need to verify the correctness of the drone's telemetry doing odometric calculations using the recorded sensory data, which requires these same parameters. The last but not the least purpose of the camera calibration was to get rid of the lens distortion. Since *GoPro* uses a fisheye lens, the image taken is not rectangular and is distorted as depicted by figure 4.2. Later on, it is going to be made clear that the mapping calculations, which assume the viewed area is rectangular, would not be correct in all parts of the image otherwise.

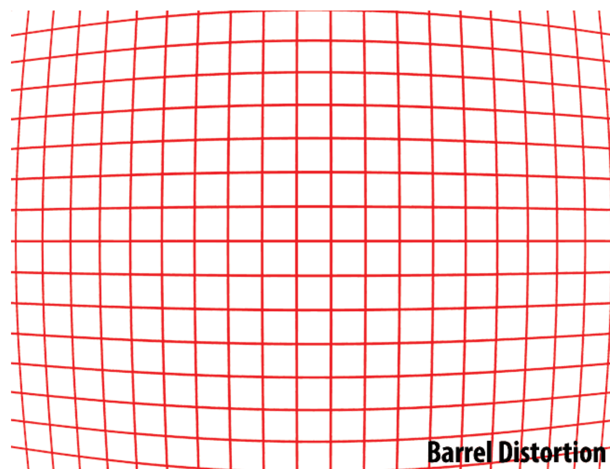


Figure 4.2. Barrel distortion.

All of the camera calibration software that was used, uses *OpenCV*'s methods. *OpenCV* is an open source computer vision library released under the BSD license. It is written in *C++*, however, it provides wrappers in other languages such as *Python*

or *Java*. As *GoPro* natively does not allow for video streaming in the correct resolution, it is necessary to capture the camera calibration data first and then use them with the appropriate calibration software.

Several approaches were tried, but due to *GoPro*'s significant distortion often yielded incorrect results. Finally, a correct calibration was performed using Camera Calibration Toolbox for Matlab [23].

4.2.1 Calibration procedure

Geometric camera calibration estimates the parameters of lens and image sensor of a camera. These parameters can be used to correct for lens distortion and perform other operations with the scene.

The calibration itself is generally done by capturing images of a chessboard, which is positioned towards the camera at different angles in every image. The calibration software then extracts the chessboard's corners and tries to estimate the camera's parameters based on the images' distortion.

Camera Calibration Toolbox for Matlab first loads the chessboard images. A naming convention should be followed and the chessboard images should be of the same format. They should be numbered and, as it was noted during this work's calibration, the numbers should not be too high with gaps between them, otherwise the performance is decreased significantly. For instance, having numbered the images 1, 359, 578, ..., 2580, 2960 made the procedure take hours, while numbering them from 1 to 20 took up to several minutes.

By running *calib_gui.m* file and following the toolbox's tutorial [23], the images are loaded into memory and the procedure begins. First, it is necessary to manually select the inner grid corners as depicted in figure 4.3.

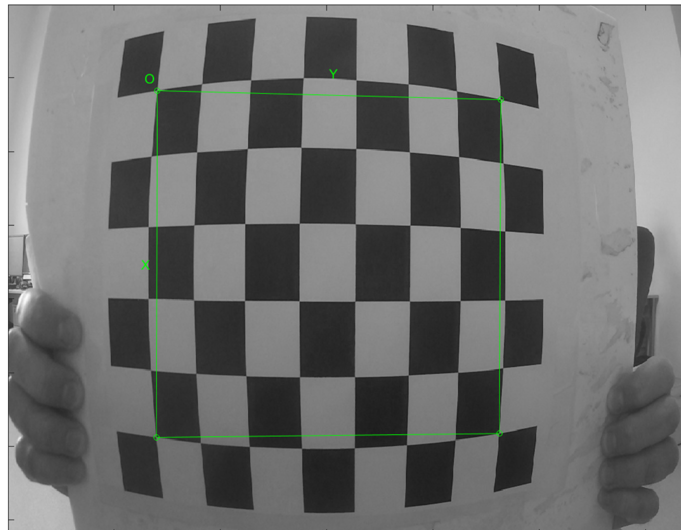


Figure 4.3. Extracted corners.

After the grid corner extraction of each image, it is necessary to guess the radial distortion factor. This operation helps the toolbox find the chessboard corners' positions and it is simply a matter of guessing. Depending on the lens distortion type, it is either

a positive or a negative number. In this work, the coefficient ranged between 0 and -1.45. Figure 4.4 shows how the grid corners are positioned before the estimate while figure 4.5 depicts the corners afterwards.

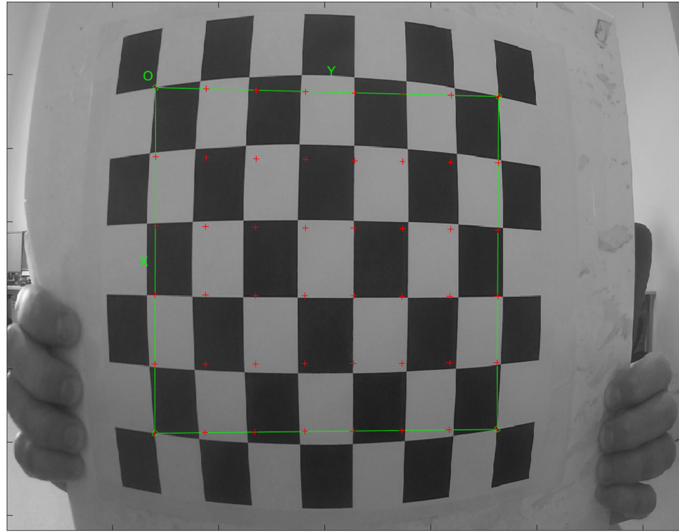


Figure 4.4. Before radial distortion factor estimate.

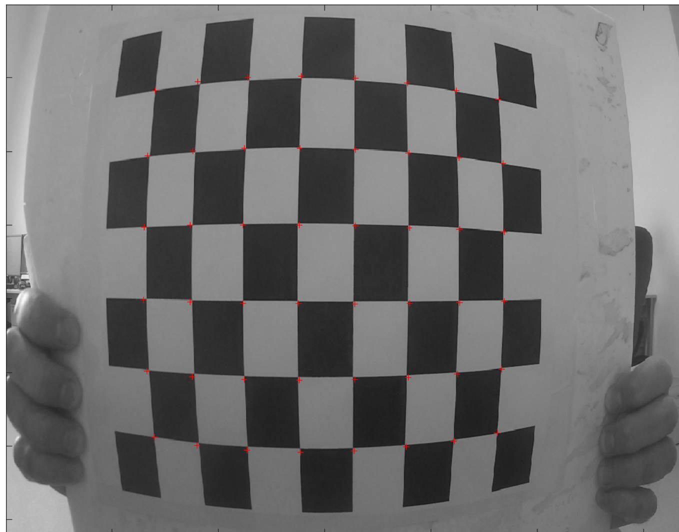


Figure 4.5. After radial distortion factor estimate.

The calibration software then finds the grid corners more precisely, however, it is important to try to make the initial estimate as precise as possible, otherwise the automatic corner extraction might fail. Once all the grid corners have been extracted, we run the camera calibration procedure and get the camera calibration matrix. For each of our images we can display the reprojection error seen in figure 4.6; if its value is too high, we can perform the corner extraction for particular images again, tune the corner extraction parameters accordingly and run the camera calibration again.

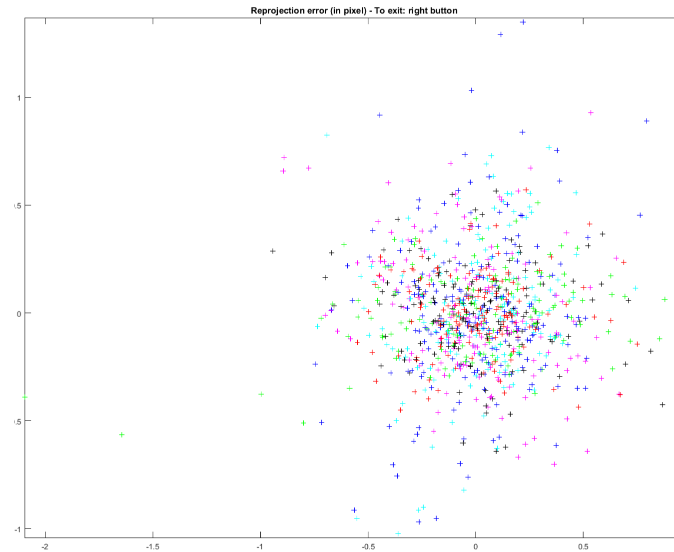


Figure 4.6. Reprojection error in pixels.

Initially, this matrix includes solely 4 distortion coefficients, thereby if we want to compute all 5 that can be computed, it is necessary to activate them by executing:

```
est_dist = [1;1;1;1;1];
```

In a similar way, some of the distortion coefficient might be rejected from the calculation.

4.2.2 Calibration results

Table 4.1 represents the resulting calibration matrix of the *GoPro* camera at ultra wide viewing angles and resolution of 1280/720 pixels. The resulting focal length and the principal point of view are then expressed in pixels. Since the center of such a resolution is at pixel 640/360, it means the principal point of view is 22 pixels towards the top and 23 pixels to the right from the center of the image.

Parameter	Value
Focal Length	[582.18394 582.52915] ± [0.77471 0.78080]
Principal Point of View	[663.50655 378.74541] ± [1.40781 1.13965]
Skew	[-0.00028] ± [0.00056] =>
	angle of pixel axes = 90.01599 ± 0.03208 degrees
Distortion	[-0.25722 0.09022 -0.00060 0.00009 -0.01662]
	± [0.00228 0.00276 0.00020 0.00018 0.00098]
Pixel error	[0.30001 0.28188]

Table 4.1. Calibration matrix.

Having calculated the distortion coefficients, it is now possible to rectify our images and compute their field of view. That way, we eliminate any possible issues with image distortion inaccuracy as stated earlier. Figures 4.7 and 4.8 demonstrate the difference of an image before and after being undistorted.

It is notable, as the image is rectified, that parts of it around its edges get cropped in order for the image to maintain a rectangular shape.

Figure 4.7. Distorted image.

Figure 4.8. Undistorted image.

Parameter	Value
FOVh	95.41677635378488
FOVh	63.43170132212425

Table 4.2. FOV of a rectified image.

Using *OpenCV*'s functions, we then calculate the viewing angles of the rectified image, as displayed in table 4.2.

In order to verify these results, an experiment described in section 5.2 was performed.

4.3 Odometric calculations

In order to verify the correctness of our drone's sensors, odometric calculations were used. The goal was to be able to take two consecutive video frames and estimate the camera's movement.

OpenCV provides such methods, however, calculating the camera's rotation and translation is a process composed of several steps:

1. Loading two images and converting them to grayscale.
2. Identifying key points of these images using the SURF (Speeded Up Robust Features) algorithm.
3. Matching these key points together.
4. Finding an essential matrix.
5. Recovering the camera's pose.

The reason why it is necessary to convert the images to grayscale is that the SURF algorithm is designed for 8-bit grayscale images for the sake of performance [24].

4.3.1 Feature detection

In order to perform the previously mentioned operations with two images, it is necessary to find common points in two images also referred to as *key points*. These key points are selected independently at distinctive locations in the image, such as corners, blobs and T-junctions. It is necessary that finding such key points is a repeatable operation under different conditions. The neighborhood of every such point is described by a feature vector, which has to be distinctive and at the same time robust to noise or geometric and photometric deformations [24].

As the SURF algorithm was removed from *OpenCV*, which can be downloaded on the official website, and was moved to its extra modules, it is necessary to do a custom *OpenCV* compilation with these extra modules. Using CMake, we specify the required compiler and the directory with *OpenCV*'s source files. Running *Configure* enables us to set additional build properties where it is important to set the *OPENCV_EXTRA_MODULES_PATH* variable to the directory containing the extra modules (natively *opencv_contrib\modules*). On Windows, it is required to set the *Java_HOME* environment variable correctly and install *Apache Ant* properly. Also in order for the compilation to run successfully, it was necessary to disable to following:

- BUILD_SHARED_LIBS
- BUILD_opencv_aruco
- BUILD_opencv_xphoto
- WITH_WEBP
- WITH_PTHREADS_PF

Also, *Python* needs to be installed in order to generate the *Java OpenCV* wrapper. Afterwards, the compilation was run successfully and the SURF algorithm ready to use. Figure 4.9 shows features detected in an image.

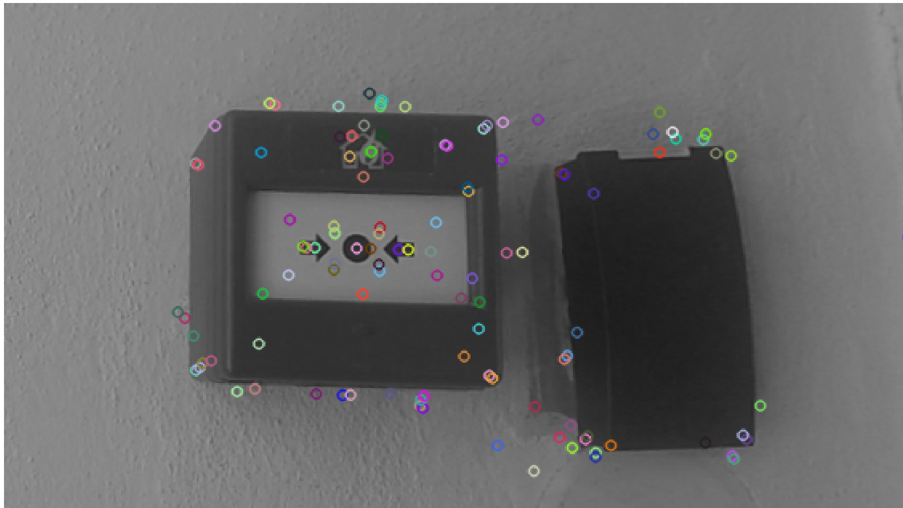


Figure 4.9. Feature detection.

4.3.2 Feature matching

Matching features of one image with features of another image was done by the *OpenCV*'s Brute-Force matcher. The algorithm takes a feature from one set and computes the distance using its distance calculator with all features from the other set. The closest feature is then returned [25]. Feature matching does not care whether the matched features are matched correctly, it just matches the two most probable pairs according to its distance calculator implementation.

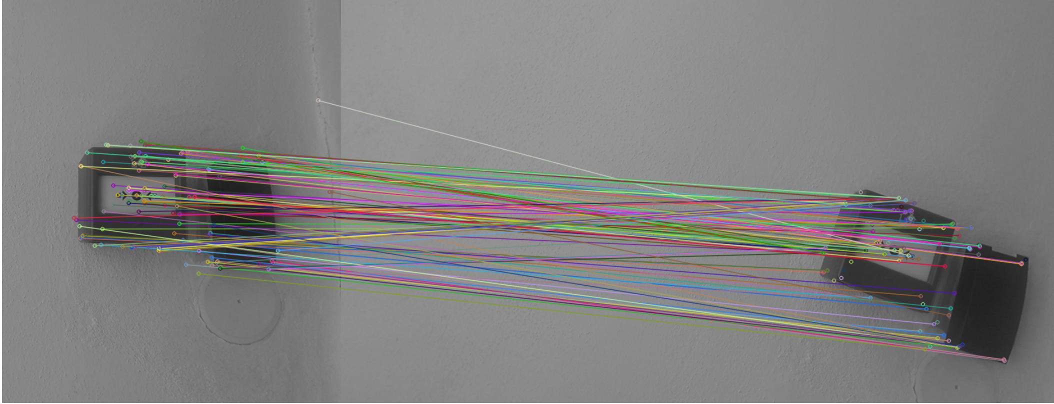


Figure 4.10. Feature matching.

Figure 4.10 shows how different key points are matched together and we can see that some of the key points were matched incorrectly. In the following section, it is explained which algorithms are used to filter out these incorrect matches.

4.3.3 Essential matrix

An essential matrix can be seen as a 3x3 matrix establishing constraints between two sets of matching image points that can be calculated in a calibrated environment. It encapsulates both the intrinsic (optical center, focal length) and the extrinsic (camera's location in the 3-d scene) parameters of the camera. Thereby, calculating and decomposing this matrix allows us to identify the relative position of a camera (or two cameras) between two images.

OpenCV provides us with means of computing such a matrix. In order to discriminate between good matches (inliers) and bad matches (outliers) from the previously matched set of key points, we can specify a method used for such a discrimination. *OpenCV* offers two methods - LMedS (Least Median of Squares) and RANSAC (Random sample consensus), which was used for our calculations.

Afterwards, we can recover the camera's pose and get its relative position by decomposing the essential matrix using *OpenCV*'s functions; such a decomposition then yields a rotation (1) and a translation matrix describing the relative position of the camera between the two images. In order to get rotations in axes x, y, z, we perform the following calculations.

$$R = \begin{Bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{Bmatrix} \tag{1}$$

Using *Java*'s **Math.atan2()** function we compute camera's roll (2), pitch (3) and yaw (4)[1].

$$roll = \arctan_2(a_{10}, a_{00}) \tag{2}$$

$$pitch = \arctan_2(a_{12}, a_{22}) \tag{3}$$

[1] These rotations are explained in section 4.5.

$$yaw = \arctan_2(a_{20}, \sqrt{a_{21}^2 + a_{22}^2}) \tag{4}$$

At this moment, all the tools necessary for data verification were ready to use and various experiments could be performed.

4.4 Data mapping

One of the first tasks of this work was to define an approach of how to describe the recorded sensory data and store them in a database. The general idea is to calculate the area being captured described by a geometrical shape based on the drone's attitude and its camera's parameters. Since the coordinates of the camera are expressed in the geographic coordinate system (GPS), and as it is in this work's context due to real time performance as well as *PostGIS'* API often convenient to use Cartesian coordinates, it is necessary to perform a coordinate conversion beforehand.

As depicted in figure 4.11, it is necessary to do a transformation from a spherical system defined by axes X, Y, Z and origin O_E to a coordinate system defined by axes x, y, z and origin O_{GS}. In our calculations, we consider Earth to be a perfect sphere.

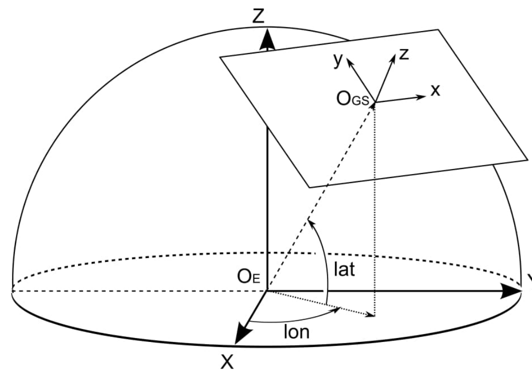


Figure 4.11. Coordinate system transformation [26].

Converting any other arbitrary point requires O_{GS} in order to be performed correctly. However, due to the curvature of Earth, the farther away the converted points are from O_{GS} the more significant the conversion error is as depicted in figure 4.12.

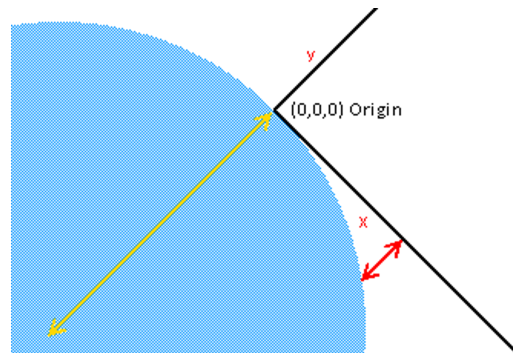


Figure 4.12. Curvature of Earth error.

Table 4.3 shows how the error differs with increasing distance considering Earth to be a perfect sphere with a radius of 6371 kilometers.

Distances	Altitude difference
100 m	0.00078 m
1000 m	0.078 m
10000 m	7.84 m

Table 4.3. Elevation error.

4.5 Mapping algorithm

In this section, the main algorithm further referred to as *mapping algorithm* which maps the collected data to the database will be described and the reader will be familiarized with the entire process in detail.

4.5.1 Bounding polygon

In order to describe the sensory data taken, basic algebraic approaches were used. We can assign the following camera's attributes to each frame:

- latitude
- longitude
- altitude
- roll
- pitch
- yaw
- vertical field of view (FOVv)
- horizontal field of view (FOVh)

The whole visible area being captured by the camera can be described by a frustum as depicted by blue lines in figure 3.7. For our algorithm it is necessary to calculate the projection of this frustum on the ground. That is done by finding the base of the frustum and connecting it to the camera's position omitting its altitude. In order to find the bottom corners of this frustum on the ground, we create 4 rays describing its sides and we look for intersections with the ground approximated by a plane. Vectors (5) define these rays facing down along the z-axis.

$$\begin{aligned}
ray1 &= (\tan(FOVh/2),\ \tan(FOVv/2),\ -1) \\
ray2 &= (\tan(FOVh/2),\ -\tan(FOVv/2),\ -1) \\
ray3 &= (-\tan(FOVh/2),\ -\tan(FOVv/2),\ -1) \\
ray4 &= (-\tan(FOVh/2),\ \tan(FOVv/2),\ -1) \qquad (5)
\end{aligned}$$

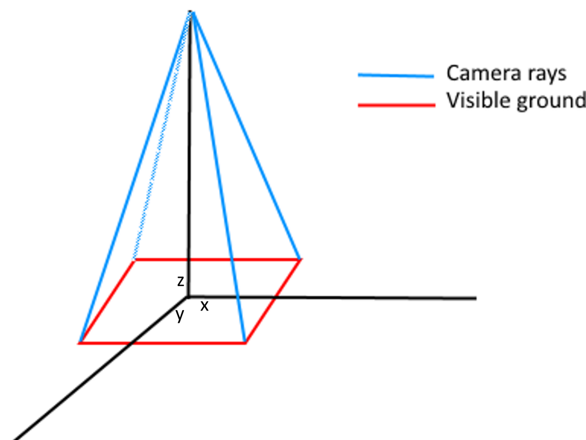


Figure 4.13. Rays constructed by (5).

Before we look for intersections with the ground, it is necessary to rotate the rays according to the camera's parameters. A yaw α is a rotation about the z-axis that can be described by a rotation matrix (6).

$$R_z(\alpha) = \left\{ \begin{matrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{matrix} \right\} \tag{6}$$

A pitch β is a rotation about the y-axis described by a rotation matrix (7).

$$R_y(\beta) = \left\{ \begin{matrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{matrix} \right\} \tag{7}$$

A roll γ is a rotation about the x-axis described by a rotation matrix (8).

$$R_x(\gamma) = \left\{ \begin{matrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{matrix} \right\} \tag{8}$$

Each of these matrices is an extension of a 2-dimensional rotation matrix, essentially performing a rotation with respect to two coordinates while leaving the third one unchanged. A single rotation matrix (9) can be formed by multiplying the yaw, pitch and roll rotation matrices. $R(\alpha, \beta, \gamma)$ performs the roll, pitch and yaw rotations respectively [27].

$$R(\alpha, \beta, \gamma) =$$
$$= \left\{ \begin{matrix} \cos(\alpha)\cos(\beta) & \cos(\alpha)\sin(\beta)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\sin(\beta)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ -\sin(\beta) & \cos(\beta)\sin(\gamma) & \cos(\beta)\cos(\gamma) \end{matrix} \right\} \tag{9}$$

Using the final rotation matrix (9), we can get the real frustum defining the area visible by the camera given by the rotated ray-vectors. Figure 4.14 depicts what such a rotated frustum looks like and it can be seen how the original rectangle is distorted.

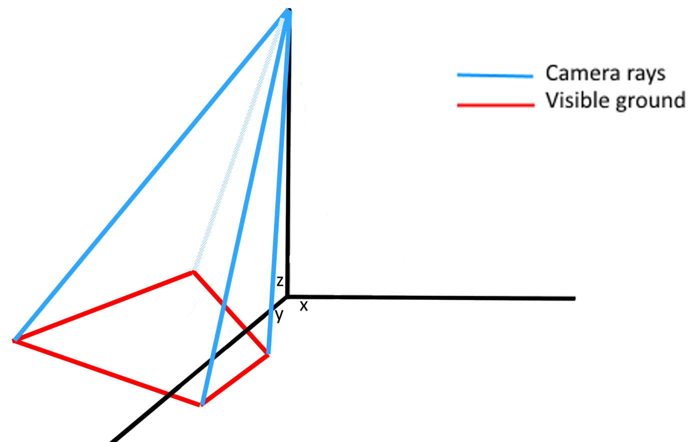


Figure 4.14. Rays constructed by (5) and rotated by (9) in all three axes.

Using these vectors, we can find the points of intersection with the ground approximated by a plane $-z = 0$. The code numbered 4.15 describes how such a calculation is performed. The parameter *Vector3d origin* represents the camera's position in Cartesian coordinates.

```
public static Vector3d findRayGroundIntersection(Vector3d ray,
 Vector3d origin) {

// Parametric form of an equation
// P = origin + vector * t
Vector2d x = new Vector2d(origin.x,ray.x);
Vector2d y = new Vector2d(origin.y,ray.y);
Vector2d z = new Vector2d(origin.z,ray.z);

// Equation of the horizontal plane (ground)
// -z = 0

// Calculate t by substituting z
double t = - (z.x / z.y);

// Substitute t in the original parametric equations
// to get points of intersection
return new Vector3d(x.x + x.y * t, y.x + y.y * t, z.x + z.y * t);
}
```

Figure 4.15. Ray-ground intersection calculation written in Java.

Having calculated the base of the frustum, we have to complete the polygon on the ground bounding the camera's view. That is done by taking the points of intersection along with the camera's position with zero altitude and constructing a convex hull. A convex hull of a set of points is the smallest convex polygon for which each point of this set lies within its interior or lies on its edge [28]. An example of a convex hull can be seen in figure 4.16.

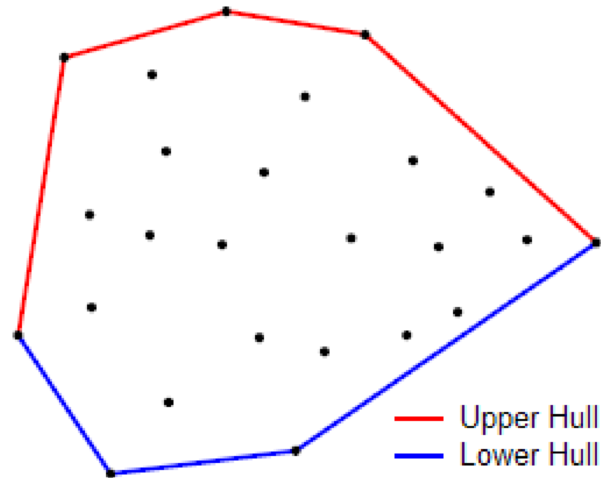


Figure 4.16. Convex hull example [29].

A visualizer was written for the purposes of development to help both to visualize the problematics and to verify various assumptions. Figure 4.17 visualizes the whole frustum with its base painted in black. In figure 4.18, we can then see the bounding polygon (painted in red) of a possible field of view.

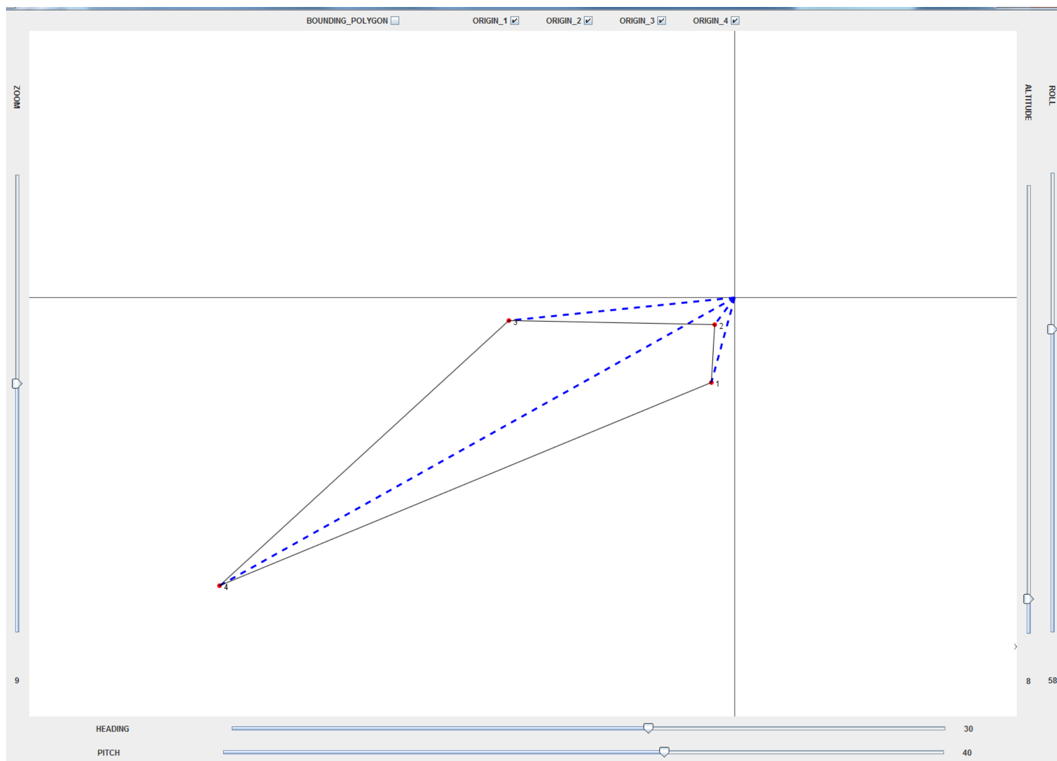


Figure 4.17. Frustum describing the camera's visible area.

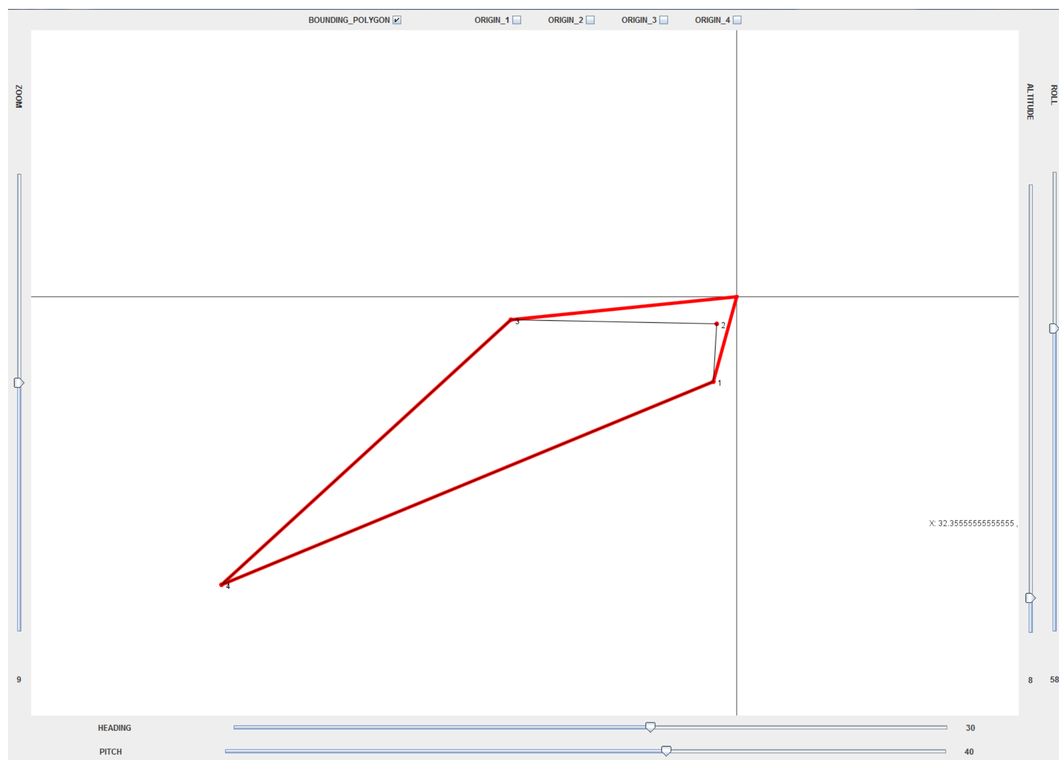


Figure 4.18. Bounding polygon.

4.5.2 Limiting the distance of view

Since the camera can be looking *over the horizon* and thus the rays defining the viewed area would not intersect the plane describing the ground correctly, there was a need to limit the distance of view in such a case. As all the previously mentioned calculations are done at the center of the coordinate system as well as the origin of the ray-vectors is $(0, 0, z)$ where z is the camera's altitude, we can expect the point of intersection with the ground to lie in the same quadrant as the direction of its corresponding ray-vector. Thereby, if the point of intersection lies in a different quadrant, we detect that the camera is looking *over the horizon*. In order to limit the range, we define a variable d_{max} being the maximum distance of view. Using equations (10) we calculate the maximum distance, ergo a corner of our bounding polygon.

$$
\begin{aligned}
x &= x_{ray} * d_{max} \\
y &= y_{ray} * d_{max}
\end{aligned}
\tag{10}
$$

It is subject to further studies to optimize this variable d_{max} as it is dependent on many factors such as the camera's resolution, current weather conditions or the illumination of the surrounding environment. For the purposes of this work, a fixed constant was used as d_{max}.

4.5.3 Translation of the bounding polygon

So far, all calculations have been done in the coordinate system's origin instead of the camera's position and it is necessary to adjust the results accordingly. Once all other operations have been done, the resulting bounding polygon needs to be translated to its real position. That is done by adding the camera's Cartesian coordinates to each of the bounding polygon's corners.

4.5.4 Additional filtering

To allow for additional data filtering and speeding up the search of a point of interest, it was decided to form bounding polygons of all monitored areas. This way while looking for a point of interest, it is not necessary to go through all data sets (picture sets, video frames) in the database but only to check those associated with the monitored areas possibly containing it. *PostGIS'* index allows for a quick look up of these areas and thus quickly filtering out unnecessary data sets. As for the whole monitored areas' bounding polygons, a different approach was used than for each frame's bounding polygon calculation described previously. Finding the right monitored area of a point of interest one is trying to seek out in frames and images requires the monitored areas' bounding polygons to be stored in the GPS coordinates. Only once the corresponding monitored area has been found, it can be converted into its Cartesian coordinates. Thereby, since we need to store these bounding polygons in spherical coordinates (GPS), the ray-plane intersection approach would not work.

The first approach to deal with this issue was to use the coordinates of all frames' bounding polygons in Cartesian coordinates, calculate a convex hull and convert its coordinates back to GPS. As each of these conversions introduces a slight error, the resulting convex hull rendered to be inaccurate, sometimes being smaller that it should be and thus possibly making a search of a point of interest impossible; thereby this approach was abandoned.

Afterwards, another approach that proved to be functional was selected. As we calculate bounding polygons of each frame, we take the camera's GPS coordinates and using an external library [30] we compute two other coordinates defining a square which represents a bounding polygon of the maximum possible viewed distance given by a constant d_{max}. Formally it is an array of two elements such as

- The latitude of any point within the specified distance d_{max} is greater or equal to the latitude of the first array element and smaller or equal to the latitude of the second array element.
- If the longitude of the first array element is smaller or equal to the longitude of the second element, then the longitude of any point within the specified distance d_{max} is greater or equal to the longitude of the first array element and smaller or equal to the longitude of the second array element.
- If the longitude of the first array element is greater than the longitude of the second element (this is the case if the 180th meridian is within the distance d_{max}), then the longitude of any point within the specified distance d_{max} is greater or equal to the longitude of the first array element or smaller or equal to the longitude of the second array element [30].

In other words, using these two new coordinates, we can form a bounding polygon of a circle limiting our field of view defined by the previously mentioned constant d_{max}. We continuously compute these bounding polygons, add their corners to a list and in the end compute a convex hull defining the whole monitored area's bounding polygon. It is important to include points of a previously computed monitored areas' bounding polygon if such a polygon exists. Figure 4.19 depicts a situation where 4 images were taken in a monitored area, their maximum possible viewed radii along with the radii's bounding polygons and the convex hull - a bounding polygon - of the whole monitored area.

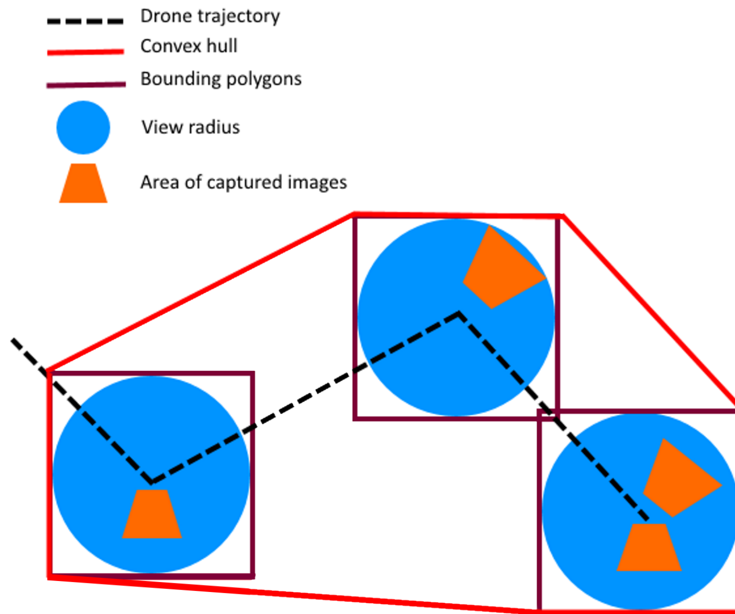


Figure 4.19. Monitored area - convex hull.

This approach is not optimal as extra space is being included in the bounding polygon. However, its main purpose is to filter out monitored areas definitely not containing a point of interest, which would only be an issue in a case where this extra space in the bounding polygon would intersect one of another monitored area. In such a case the search time would increase.

A more precise approach would be, for instance, to calculate points of intersection, as it is done with a plane representing the ground in Cartesian coordinates, using spherical geometry. However, due to time constraints such an approach was not tested.

4.5.5 3-dimensional check

Up until now the algorithm was solely trying to discriminate between data definitely not containing a point of interest and data possibly containing it. As a result, verifying whether a point of interest lies within a particular set of images can be performed faster since the data set we work with now is significantly smaller.

The 3-dimensional check uses basic algebraic approaches; first we compute the equations of planes defining the camera's viewed area neglecting it is a frustum but considering it as a pyramid. That is done by using the previously defined rays (5) always taking a neighboring pair and calculating their cross product resulting in a norm vector of one of the pyramid's sides. Afterwards, we compute the dot product of each plane's normal with the point of interest, which yields the distance from the plane and the point. If the result is in all 4 cases positive, the point of interest lies within the pyramid, otherwise it lies outside. This approach is visualized by figure 4.20.

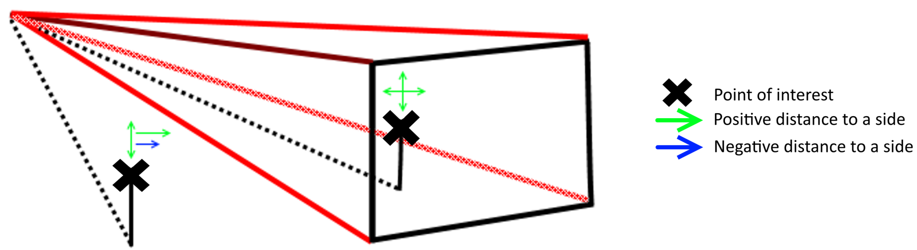


Figure 4.20. 3-dimensional check example.

4.6 Database structure

As mentioned before, the database used in this work is *PostgreSQL* with its spatial extension *PostGIS*. In this section we will discuss and describe the database model used. The requirements for the database were to be capable of storing information both about videos and image sets taken by a camera mounted on a drone. It needs to store everything required for this project's necessary calculations as well as enable quick data search and scalability. At the same time it needs to be able to avoid data redundancy creation and thus possibly preventing insertion, deletion and modification anomalies. This process is achieved through database normalization. Figure 4.21 depicts the database relational schema.

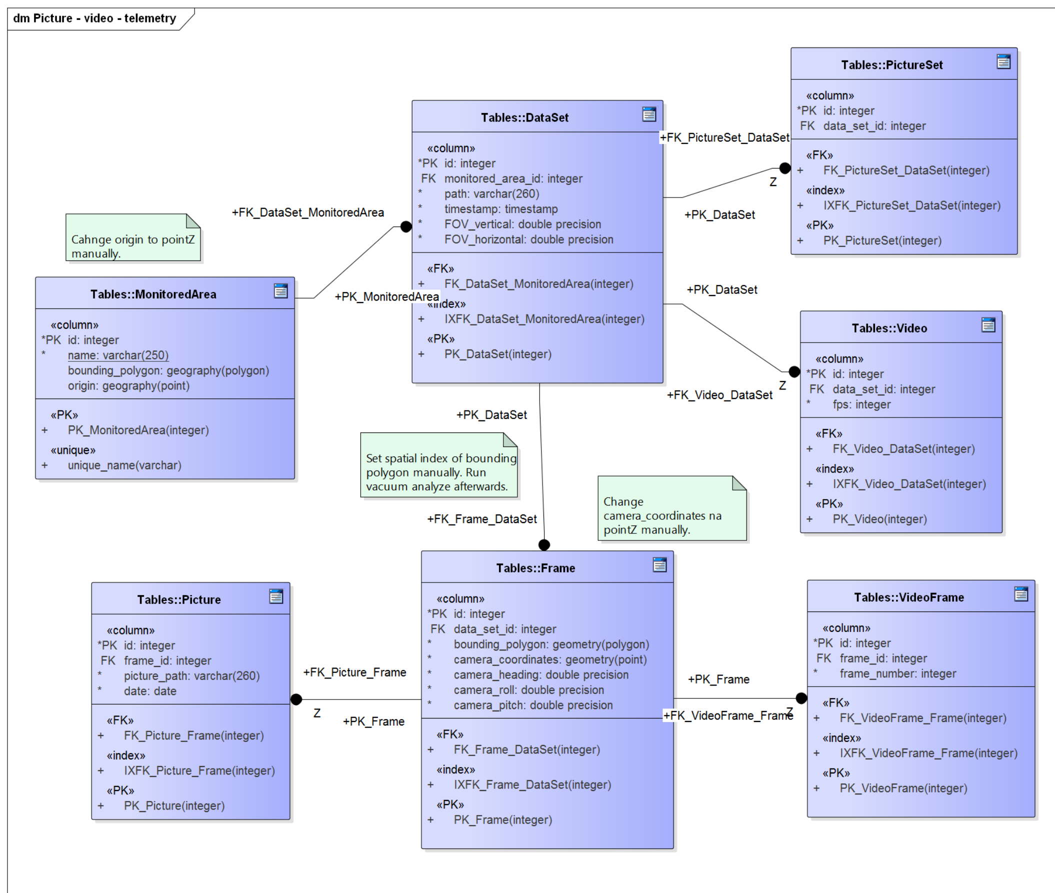


Figure 4.21. Database model.

The main entity is called *MonitoredArea*. It represents an area, object or a set of objects that lie in a geographical proximity. It is defined by the following attributes:

- **id:** The primary key of each MonitoredArea that is generated automatically.
- **name:** A unique name containing up to 250 characters.
- **bounding_polygon:** Bounding polygon of the entire area in GPS.
- **origin:** Point used for coordinate conversion from GPS to Cartesian coordinates and back. By convention, it is the first coordinate from the first dataset saved in the database that is related to the corresponding MonitoredArea. While building create statements, it is important to change the data type of **origin** from *point* to *pointZ*.

DataSet represents a particular data collection. The IS-A database approach [22] enables us to store the information about picture sets or videos in the database using the same base entity and is similar to object inheritance in object oriented programming. The benefit of this approach is that it prevents us from introducing redundant data to the database.

- **id:** The primary key of a data set that is generated automatically.
- **monitored_area_id:** The foreign key identifying the data set's corresponding monitored area.
- **path:** A path to a video file or a directory containing images.
- **timestamp:** Defines the time when the data set was created in Epoch time format. In other words, *timestamp* is the number of milliseconds that have elapsed since 1 January 1970.
- **FOV_vertical:** Vertical field of view of the camera used to capture this data set.
- **FOV_horizontal:** Horizontal field of view of the camera used to capture this data set.

One of two tables specifying the type of a data set is *PictureSet* defining that a *DataSet* is a set of images.

- **id:** The primary key that is generated automatically.
- **data_set_id:** The foreign key identifying the corresponding *DataSet*.

And similarly *Video* defining that a *DataSet* is a video.

- **id:** The primary key that is generated automatically.
- **data_set_id:** The foreign key identifying the corresponding *DataSet*.
- **fps:** Frames per second of the video.

Frame represents either one frame of a video in case of a video set or one image in case of a picture set using the IS-A approach again [22].

- **id:** The primary key of a frame that is generated automatically.
- **data_set_id:** The foreign key identifying the corresponding DataSet.
- **bounding_polygon:** Bounding polygon of the area viewed by the camera on the ground in Cartesian coordinates.
- **camera_coordinates:** Camera's Cartesian coordinates.
- **camera_heading:** Camera's heading (yaw) in radians.
- **camera_roll:** Camera's roll in radians.
- **camera_pitch:** Camera's pitch in radians.

Similarly to *DataSet*, there are two types of *Frame.Picture* defines a picture set element.

- **id:** The frame's primary key that is generated automatically.

- **frame_id:** The foreign key identifying the corresponding *Frame*.
- **picute_path:** The picture's path on disk.
- **date:** Date of the picture's creation.

The other entity describing a video frame is *VideoFrame*.

- **id:** The frame's primary key that is generated automatically.
- **frame_id:** The foreign key identifying the corresponding *Frame*.
- **frame_number:** Frame's number in the video.

4.6.1 Additional adjustments

As only constrained columns such as primary keys or unique columns are indexed by default, it is necessary to add index on columns manually when required. That is done by command:

```
 CREATE INDEX [index_name]
 ON table_name
 USING GIST ([column_name]);
```

A particular example from this work would then be:

```
CREATE INDEX bounding_polygon_gix
ON "Frame"
USING GIST (bounding_polygon);
```

It is important to specify the *GIST* index type, as by default *PostgreSQL* uses B-Tree indices, which are not lossy and take up significantly more space than *GIST* indices that index only the bounding box of a particular geometry.
After an index has been created, it is important to run:

```
VACUUM ANALYZE [table_name];
```

in our case:

```
VACUUM ANALYZE "Frame";
```

The **ANALYZE** command updates the internal database's statistics and **VACUUM** asks *PostgreSQL* to reclaim any unused space in the table pages left by updates or deletes to records. When adding an index, **VACUUM** is necessary for the database to run effectively. *PostgeSQL* provides an `autovacuum` option that is enabled by default that runs **VACUUM ANALYZE** automatically at sensible intervals. However, it might not be a good idea to wait for autovacuum after adding new indices or a big amount of data [12].

One last thing to note, which is not captured by the diagram 4.21, is that column *origin* in table *MonitoredArea* and column *camera_coordinates* in table *Frame* need to be changed from type *point*, which allows only for the x and y coordinates, to type *pointZ*, which also adds the z coordinate. The diagram modeler, however, did not allow for this type.

At this point, it was possible to perform the first experiment that is described in section 5.1.

Chapter 5
Experiments

This chapter will go through experiments implemented throughout the development of this work chronologically and it will describe their motivation, implementation and their results.

5.1 Visual experiment

The context of this work's two visual experiments were fields with several distributed objects of known GPS coordinates. A previously described drone with a mounted *GoPro* camera was used. The experiment was focusing on both objects with zero altitude and objects at different heights with the camera both facing down and slightly forward and it was using *GoPro*'s ultra wide field of view. At the time of both experiments the weather was windy which notably affected the drone's behavior.

A short footage of several minutes was taken with the drone hovering randomly over the fields with the distributed objects and its telemetry was recorded. Then, the mapping algorithm was used to calculate the necessary data and to map them to the video footage.

Two approaches were used to verify the correctness of the mapping algorithm. The first one consisted of writing a data visualizer in *Java*, which would query the database for the camera's viewed area on the ground and visualize it along with the distributed objects on the ground. Simultaneously, the recorded video would be played and it would be observed whether in fact the calculated data corresponds to reality. The visualizer is depicted in figure 5.1 and its corresponding video frame is shown next to it in figure 5.2.

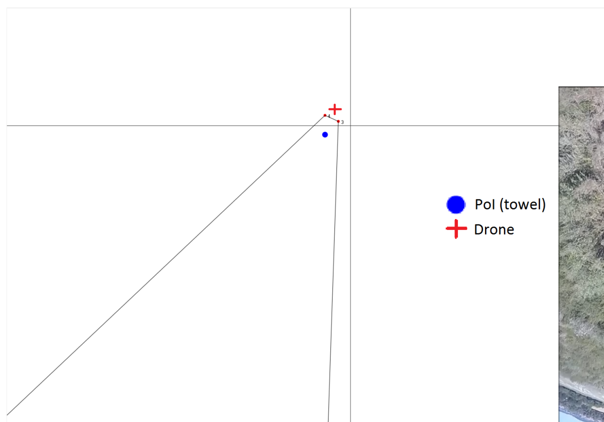


Figure 5.1. Calculated area being captured displayed by the visualizer.

Figure 5.2. The corresponding video frame.

The other approach would query the database for all frames containing a GPS coordinate that is within the camera's field of view. Then it would be verified visually again whether the GPS position lies in the frame.

5.1.1 Visual experiment - results

The results were satisfying in case the object of interest was located close to the center of the field of view. However, there was trouble with false detection when the object of interest was located outside of the image close to its edge. Several possible causes were identified.

First off, it was necessary to make sure the telemetry and the video were aligned correctly as they both originated from independent sources. The autopilot starts recording its telemetry independently of the video since there is no trigger that would activate the autopilot and the camera at the same time. However, as the telemetry starts being recorded by the autopilot, a beep of about three seconds long can be heard. It was unsure whether the telemetry starts being sent before or after the beep, thereby both the possibilities were tested in new visual experiments; an odometric experiment described in section 5.3 was performed thereafter.

The next possible inaccuracy might have been caused by inaccurate attitude retrieved from the drone's sensors. That might happen due to different reasons, such as inconvenient weather conditions or sensor inaccuracy. In that case the resulting calculated viewed area would differ dramatically from the reality. As, for instance, a larger or a smaller area that is captured in a video frame or in an image would be calculated depending on whether the measured altitude was higher or lower than reality. Considering the image is perfectly rectangular, figure 5.3 depicts how the altitude error of one meter affects the calculated viewed area with parameters specified in table 5.1 and the camera facing down.

Parameter	Value
Altitude	8m
Roll	0°
Pitch	0°
Yaw	0°
FOVh	118.2°
FOVv	69.5°

Table 5.1. Drone's real parameters of figure 5.3.

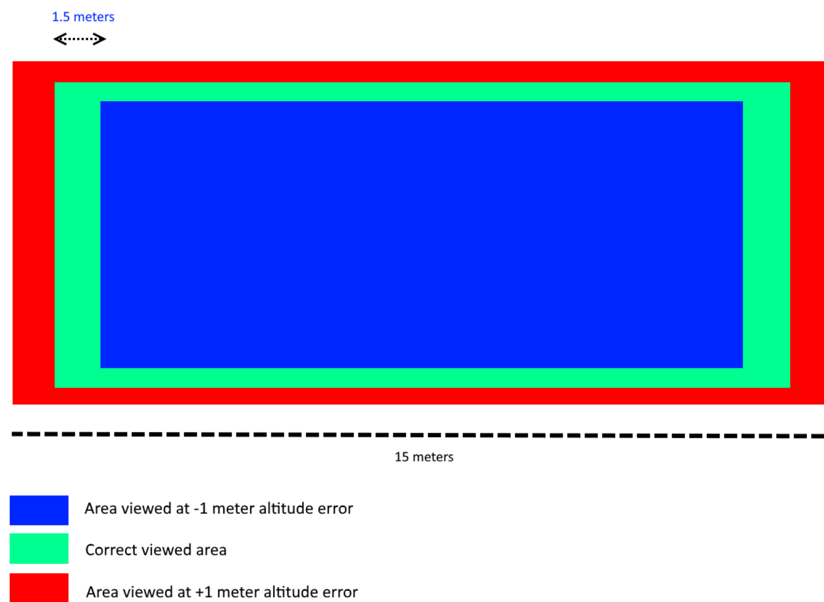


Figure 5.3. Altitude error - viewed area.

According to the autopilot's documentation, the possible and rather common altitude error might be of up to 2 meters. This might happen due to the fact that when the UAS is moving, a low pressure bubble may form on top of it leading the sensors to believe the UAS is climbing.

The last possible reason are the GPS coordinates retrieved that might differ, due to specifications, by 2 meters [17]. The offset of the GPS position would then equal to the offset of the calculated viewed area. During this work's experiments, it was observed that even though the GPS was accurate most of the time, at times it differed by several meters, which affected the results dramatically.

As it was assumed that the error might results due to any of these factors or as any of their combinations, the following step in the development was a calibration of the *GoPro* camera, which would enable us to perform odometric calculations and to rectify the camera's resulting images and verify the exact camera's parameters.

5.2 Field of view experiment

In order to verify the correctness of our camera calibration described in 4.2 a visual experiment calculating the camera's field of view was performed.

An object, in this case a piece of furniture, of a known size was captured by the camera at a known distance so that it occupied the entire length from one side of the image to the opposite one. Afterwards, using trigonometry, the viewing angles were calculated. Figure 5.4 depicts how such a calculation was performed.

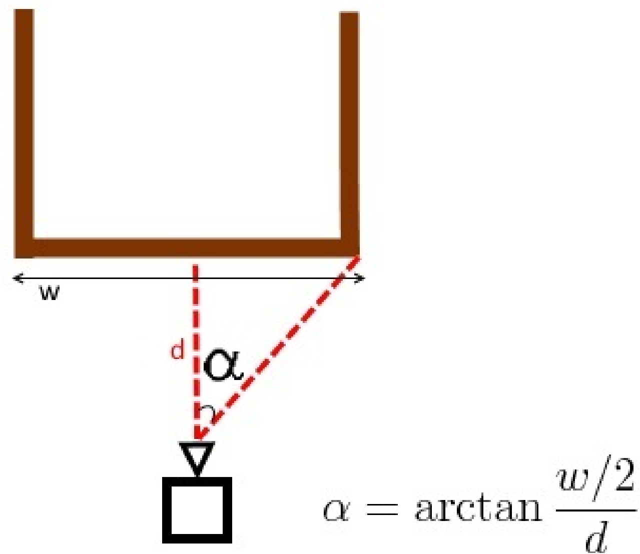


Figure 5.4. Field of view calculation.

The next step was to undistort the previously used image and identify points on the furniture that limit the image on its opposite sides. Afterwards, the same points on the furniture used previously were identified and the distance between them measured. Again, using trigonometry, the new viewing angles were calculated. Figures 5.5 and 5.6 show the images used for our calculation.

Figure 5.5. Distorted image. **Figure 5.6.** Undistorted image.

5.2.1 Field of view experiment - results

Table 5.2 shows the results of this experiment. The first column specifies the measured parameter, the second column are the official *GoPro*'s specifications of a distorted image, the third column is the result of our experiment of a distorted image, the fourth column represents the result of our measurement of an undistorted image and the final fifth column shows the output of an *OpenCV* function, which takes the camera matrix, image size and the camera's sensor dimensions as its parameters.

We can see that the results of our measurements slightly differ from *GoPro*'s specifications but this might be due to measurement errors that inevitably occur considering our limited set of tools (a measuring tape, rulers). However, for the purposes of verifying the correctness of OpenCV's algorithm, the results were concluded as satisfying.

Parameter	(D) Official spec.	(D) Measured	(U) Measured	(U) OpenCV
FOVh	118.2°	118°	96°	95.5°
FOVv	69.5°	68°	64°	63.5°

Table 5.2. Field of view experiment results.

5.3 Sensor correctness experiment

Sensor correctness experiment was done using odometric methods described in the previous chapter. Unfortunately, it was impossible to use the data which were used for the main experiments as most of the video frames are covered with grass and bushes and it is difficult for the feature matcher to match the frames' features correctly.

The experiment consisted of recording a video in a laboratory with different objects distributed around it with a camera mounted on the same drone that was used during other experiments simulating its movement in all axes while its telemetry was being recorded. Afterwards, the data was processed as usual and calculations described in 4.3 were performed. However, it was not feasible to use each two consecutive frames of the video, perform these calculations and, for instance, compute an average result of a large data set as not always the feature matcher manages to match the image's features correctly. That would happen due to relatively fast movement of the camera resulting in blurry images or because not enough identical features were captures in both of the consecutive frames. For this reason, all image pairs had to be chosen manually and then it had to be verified whether the feature matching had been successful. Movements in all 3 axes were calculated independently of each other. These results were then compared to the drone's telemetry.

Before examining the accuracy of the drone's telemetry, it was necessary to make sure it was aligned with the video correctly as described previously in 5.1.1. The idea was to compare all previously mentioned odometric calculations with the drone's telemetry assuming two scenarios; it either starts being recorded before or after the beep. First, it was compared to the time in the video corresponding to the beginning of the beep and then to its end. It turned out that the telemetry and odometry matched best to the time after the beep.

5.3.1 Sensor correctness experiment - results

Even though the data set consisted solely of 25 samples (this dataset contained subsets for roll, pitch and yaw rotations, each varying in numbers) for reasons mentioned previously, the results could provide some idea about how correct the telemetry is. Roll and Pitch never differed by more than 2 degrees, but most of the time it did not differ by more than 1 degree. Yaw, however, often yielded results very different from the reality; the differences were sometimes marginal, however, other times it differed by more than 30 degrees. The average difference from a data set consisting of 9 samples was 11 degrees.

5.4 Visual experiment revisited

With all the knowledge acquired in the previous experiments, the visual experiment described in 5.1 was performed again. This time, however, several adjustments were

made. First off, the images which were worked with were rectified and the camera's field of view used for the mapping algorithm adjusted accordingly. Then, new data sets were created, used with the mapping algorithm and then their results were compared. These data sets were identical with the original data set except for their altitude had been altered. The first new data set had its altitude increased by two meters while the other one had it decreased. This accounted for the possible altitude error given by the sensors' inaccuracy described in the autopilot's manual.

5.4.1 Visual experiment revisited - results

Figures 5.7 and 5.8 depict the new results of the visual experiments. The horizontal axis represents the video frames while the vertical axis represents different mapping algorithm altitudes for the blue, orange and green lines and whether the object of interest is currently located in the video for the red line. The green, orange and blue lines represent results of the mapping algorithm. If they are present in the chart, it means, the object of interest is present in the corresponding video frame; it is not present otherwise. All these values are arbitrary and were selected so in order to make the results transparent.

The red line titled as *Reality* was created manually observing the video frame by frame and is set to 1 if the object of interest is not visible in the video and is set to 2 if it is visible. The green line titled as *Lower* means the altitude was lowered by two meters from what had been received from the sensors while the blue line titled *Higher* means the altitude was increased by two meters. The orange line *Real* represents a computation performed with the unadjusted data.

In both the charts, it can be seen that the results are not completely accurate. Ideally, at least one of the three lines representing the mapping algorithm should overlap the red columns representing whether the object of interest truly lies in the corresponding video frame and at least one of them should not be present while the object of interest is not present; that is not, however, always the case.

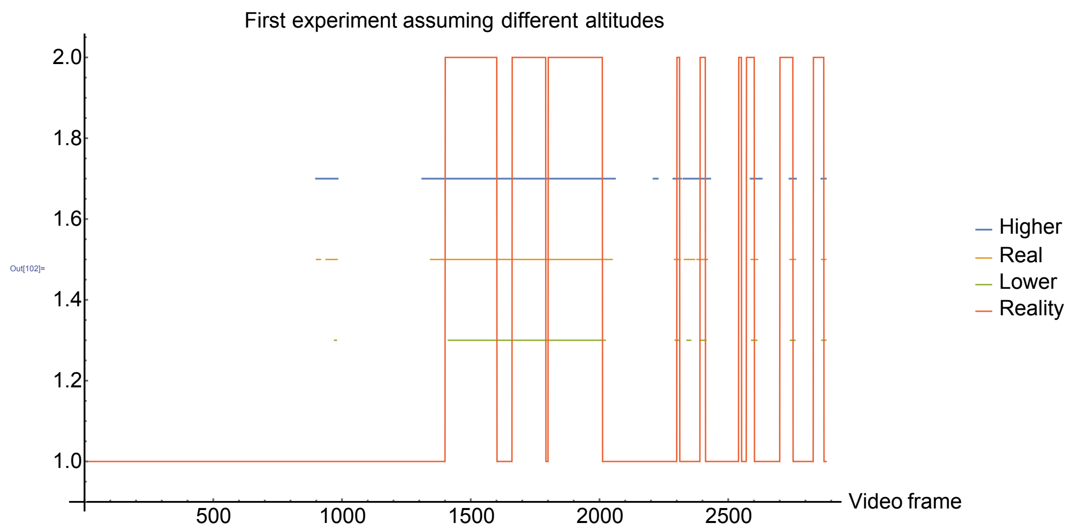


Figure 5.7. First visual experiment results.

In figure 5.7 there can be seen a false detection before frame 1000 in all the altitudes and is the least notable in the lowest one. Other false detections worth mentioning in

all altitudes are around frame 1700. In this part of the video the object of interest was located on the edge of the image and occasionally disappeared. In this case, even the slightest of all the previously mentioned inaccuracies might alter the results and it can be assumed the incorrect detection arose as their result. From frame 2250, the drone is positioned several meters away from the object of interest and starts being rotated around the vertical z axis at a relatively fast angular speed. It can be seen that the detection is off-set around the last 3 columns, which was presumed had been the result of the magnetometer's inaccuracies, but there was no way to verify this presumption.

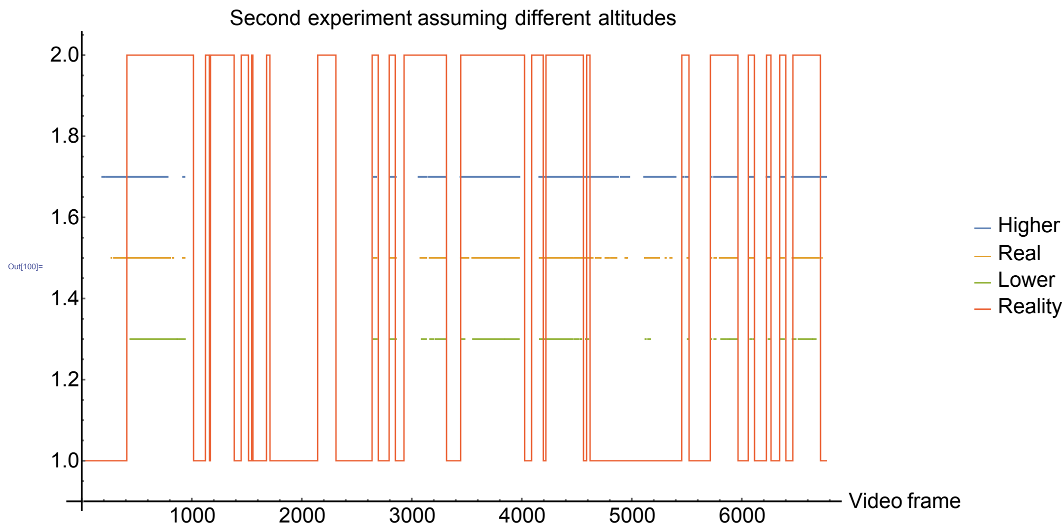


Figure 5.8. Second visual experiment results.

In figure 5.8 the results are visibly incorrect from frame 1000 until frame 2150. Having reviewed the data used for the calculations, it was found out that the GPS coordinates of the drone, which were recorded, differed dramatically (by up to 4 meters) and the camera's position was assumed to be not in front (facing the object of interest) but behind it. Eventually the GPS coordinates were corrected and except for some slight inaccuracies and false detections, the results were correct.

In the end, it seems that the most accurate results are yielded while assuming a lower altitude but due to time constraints the most precise method of object detection is yet to be determined in further works.

5.5 Scalability

An important thing to consider in this work's context was the scalability of its design. As one of the goals was to build a system enabling storing of large quantities of data and offering a good real time performance, the database and its ability to handle a growing amount of data had to be be tested. In this experiment, an entry represents meta data corresponding to a video frame or an image; therefore 10^7 entries represent 111 hours of video footage considering a frame rate of 25 frames per second.

5.5.1 Search

The time complexity of a search in an average case, considering the spatial entries do not overlap, is $O(log_M n)$. The worst case time complexity is $O(n)$ if all entries overlap

[13]. M is the maximum number of entries in a node and n is the total number of entries in the tree.

The experiment consisted of inserting different volumes of randomly generated polygons in the database, performing 1000 of random searches without indexing the spatial data and then performing the same experiment again, however, this time with a spatial index. Table 5.3 and figure 5.9 show the results of the experiment. The first column of the table expresses the number of polygons in the database and the next two columns show the average search time of one entry with a spatial index and without it respectively.

Number of polygons	Without index	With index
10^3	1 ms	0 ms
10^4	2 ms	0 ms
10^5	29 ms	0 ms
10^6	321 ms	0 ms
10^7	3722 ms	0 ms

Table 5.3. Time of search.

It is clear that spatial indexing decreases the search time significantly and even with a large amount of polygons in the database the average search time did not exceed 1 millisecond, while without the index, the search time is increasing linearly as one might expect since all entries in the database might have to be searched.

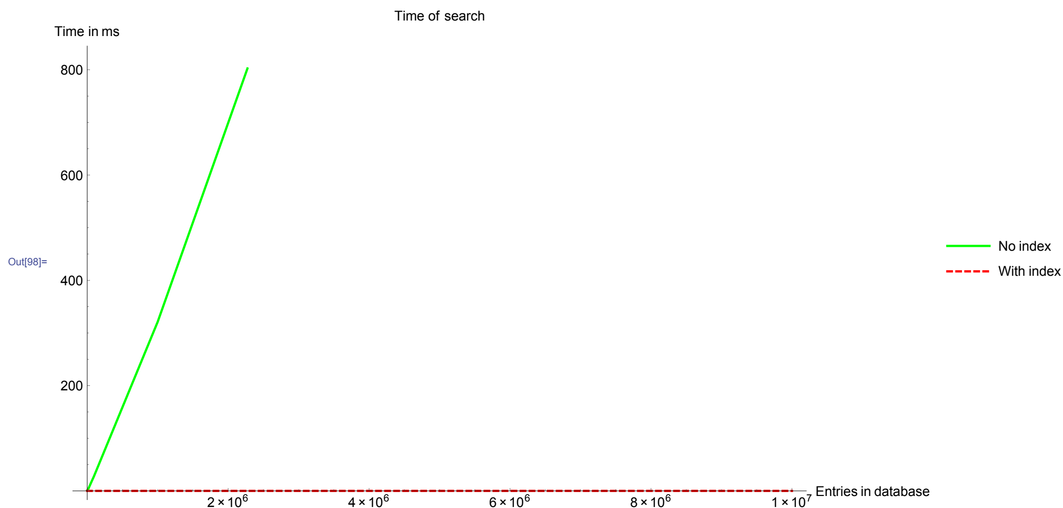


Figure 5.9. Time of search experiment.

5.5.2 Insert

The second aspect to consider was inserting new data to the database. As multiple columns of the inserted data are indexed, the index tree needs to be balanced in order to maintain its search properties. A method that is sometimes used is that before a bulk insert, we drop an index and rebuild it after the insert has finished. Table 5.4 and figure 5.10 show the results of two inserts with and without a spatial index.

Number of inserted polygons	Without index	With index
10^3	0.35 s	0.34 s
10^4	2.6 s	2.5 s
10^5	27 s	24 s
10^6	280 s	266 s
10^7	3190 s	2805 s

Table 5.4. Time of insert.

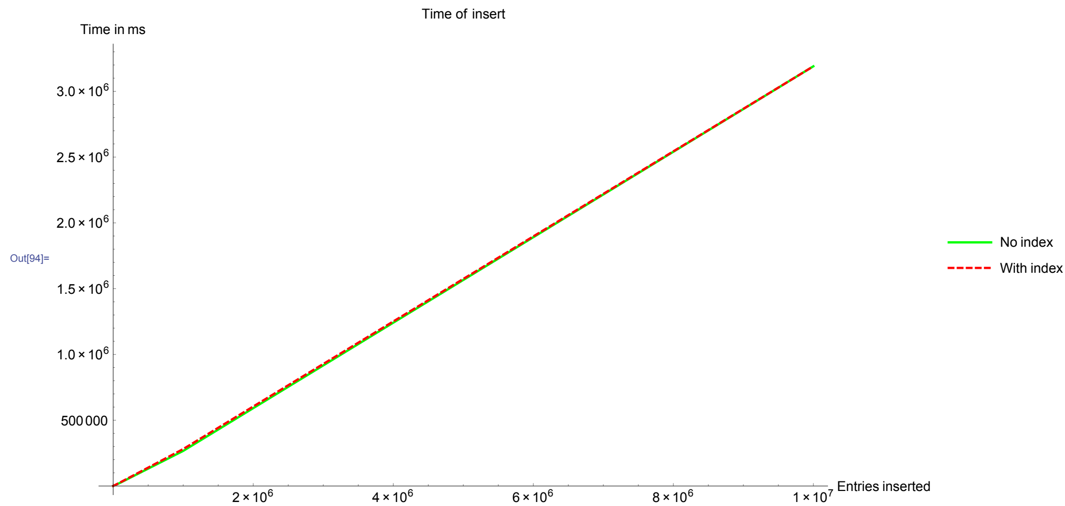


Figure 5.10. Time of insert experiment.

The results show an improvement of 6 minutes and 25 seconds of an insert into a table without the spatial index while inserting a data set of 10^7 entries. However, adding the index and running **VACUUM ANALYZE** afterwards took over 23 minutes (1380 seconds), which renders this method ineffective. Therefore dropping an index before inserting a data set of 10^7 elements and then rebuilding it does not improve the database's performance.

Chapter 6
Conclusion and future work

This thesis studied the problematics of how to georeference and how to organize sensory data acquired from Unmanned Aerial Systems and different existing commercial and non-commercial solutions that are currently being used in this area were explored. An algorithm allowing for sensory data georeferencing and for searching parts of such data containing objects of interest given by these objects' geographical position was designed and implemented. A solution of organizing and processing such data so it can be stored and worked with in an effective way was proposed. In the end, the proposed solutions were tested via experiments using real data and their results were demonstrated.

Various means of how to store geo-spatial sensory data were explored and it was found out that the *PostgreSQL* database with its *PostGIS* extension offers the most convenient environment for these purposes due to its spatial data indexing and a rich set of geo-spatial functions it possesses. The algorithm used to seek out sensory data parts containing objects of interest stores the area visible in each video frame or image in a 2-dimensional format. It takes advantage of *PostGIS*' spatial indexing and its fast 2-d operations to discriminate between data sets possibly containing an object of interest and data sets definitely not containing it. As a result, operations used for object of interest detection are done using only relevant data sets and thus are performed quickly.

Real data used for experiments were acquired using Y6 hexacopter, u-blox NEO-M8N, ArduPilot Mega 2.6 and GoPro HERO 4 Black. As for seeking out sensory data containing objects of interest, it was uncovered that the most accurate results were yielded when the camera lens' distortion was corrected as it allowed for more precise geometrical calculations. Also, the biggest inaccuracies arose from inaccurate altitude measurements and the most accurate results were rendered when considering the UAS' altitude to be 2 meters below the altitude recorded by the UAS' sensors. This most likely happens due to the fact that when the UAS is moving, a low pressure bubble may form on top of it leading the sensors to believe the UAS is climbing. Other relevant inaccuracies were caused by u-blox NEO-M8N occasionally recording incorrect GPS coordinates, or by inaccurate measurements of the UAS' compass.

Considering the results of the experiments, this work's solution can be used to organize sensory data acquired from Unmanned Aerial Systems and to seek out parts of these data containing objects of interest given by their geographical position. However, due to measurement inaccuracies, the results are not guaranteed to be correct and therefore further work is required to make the results more reliable. Other possible improvements, that have not been explored due to time constraints, which would improve the functionality of the proposed methods might include:

- A more precise way of determining the UAS' position and attitude (geographical position, roll, pitch, heading), by using more accurate devices or finding a way to detect incorrect data to be filtered out or corrected.

- Exploring a way of how to describe and store the image captured by the camera without image undistortion, allowing for larger areas being recorded.
- Deciding whether the use of *PostGIS*' geography format outperforms the geometry format as explained in section 3.2.
- Finding a way to reduce the amount of video frame meta data by filtering out similar frames reducing the size of data saved to the database and speeding up the time of search and insert.

References

[1] M. Blaha, H. Eisenbeiss, D. Grimm, and P. Limpach. *DIRECT GEOREFERENCING OF UAVS*. 2011.
http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XXXVIII-1-C22/131/2011/isprsarchives-XXXVIII-1-C22-131-2011.pdf.

[2] ICAO Cir 328. *Unmanned Aircraft Systems (UAS)*. 2011.
http://www.icao.int/Meetings/UAS/Documents/Circular%20328_en.pdf.

[3] Brian P Tice. *Unmanned Aerial Vehicles – The Force Multiplier of the 1990s*. 1990.
https://web.archive.org/web/20090724015052/http://www.airpower.maxwell.af.mil/airchronicles/apj/apj91/spr91/4spr91.htm.

[4] Sally French. *How drones will drastically transform U.S. agriculture, in one chart*. 2015.
https://en.wikipedia.org/wiki/File:Atomic_force_microscope_block_diagram.svg.

[5] Tomas Trafina. *Construction of 3D Point Clouds Using LiDAR Technology*. 2016.
https://support.dce.felk.cvut.cz/mediawiki/images/d/d3/Bp_2016_trafina_tomas.pdf.

[6] LLC H2H Associates. *Point Cloud Processing / Data Management*. 2016.
http://h2hassociates.com/services/point-cloud-processing-data-management/.

[7] Josh Elmore. *Photogrammetry*. 2016.
http://www.joshelmore.com/photogrammetry/.

[8] R. Sears, C. Ingen, and J. Gray. *To BLOB or Not To BLOB: Larger Object Storage in a Database or a Filesystem?* 2006.
https://arxiv.org/ftp/cs/papers/0701/0701168.pdf.

[9] Wiki.GIS.com. *Spatial Join*. 2016.
http://wiki.gis.com/wiki/index.php/Spatial_Join.

[10] Carrie. *MySQL spatial VS PostGIS*. 2014.
http://www.programering.com/a/MTNwQjMwATI.html.

[11] The PostgreSQL Global Development Group. *PostgreSQL*. 2016.
https://www.postgresql.org/.

[12] Mark Leslie, LISAsoft, and OpenGeo. *Introduction to PostGIS*. 2009.
http://revenant.ca/www/postgis/workshop/indexing.html.

[13] J Pei. *R-Tree, Database Systems II, Lecture slides*.
http://www.cs.sfu.ca/CourseCentral/454/jpei/slides/R-Tree.pdf.

[14] Marcelo Gattass, and Lopes Helio. *FGNG: A fast multi-dimensional growing neural gas implementation*. 2014.
https://www.researchgate.net/figure/260011432_fig2_Fig-2-An-R-Tree-data-structure-example-Adapted-from-11.

[15] IIT BOMBAY Subham, Roy. *Benchmark design for spatial database*. 2011.
https://web.archive.org/web/20120813184338/http://www.gise.cse.iitb.ac.in/wiki/images/c/c4/Finalreport.pdf.

[16] M Zajacik. *Control of a heterogeneous team of autonomous UAVs*. Master's Thesis, Faculty of Electrical Engineering, Czech Technical University in Prague. 2016.

[17] u-blox AG. *NEO-M8, u-blox M8 concurrent GNSS modules, Data Sheet*. 2015.
https://www.u-blox.com/sites/default/files/NEO-M8_DataSheet_(UBX-13003366).pdf.

[18] u-blox AG. *NEO-M8 series*. 2016.
https://www.u-blox.com/en/product/neo-m8-series.

[19] Ardupilot Mega Project. *ArduPilot Mega*. 2016.
http://www.ardupilot.co.uk/.

[20] B & H Foto & Electronics Corp. *GoPro HERO4 Black*. 2016.
https://static.bhphoto.com/images/images500x500/gopro_chdhx_401_hero4_black_edition_adventur
jpg.

[21] u-blox AG. *LEA-6, u-blox 6 GPS Modules, Data Sheet*. 2015.
https://www.u-blox.com/sites/default/files/products/documents/LEA-6_DataSheet_(UBX-14044797).pdf.

[22] J. Fong. *Mapping Extended Entity Relationship Model to Object Modeling Technique*. 1995.
http://delivery.acm.org/10.1145/220000/212007/P018.pdf?ip=147.32.83.203
&id=212007&acc=ACTIVE%20SERVICE&key=D6C3EEB3AD96C931%2E9BD1EC80ACA8C1C5%2E4D4702B0C3E38B35%2
&CFID=861014898&CFTOKEN=10594655&__acm__=1478614769_2b58eb74d41a42f5d83831f4d1d0be75.

[23] Jean-Yves Bouguet. *Camera Calibration Toolbox for Matlab*. 2015.
https://www.vision.caltech.edu/bouguetj/calib_doc/.

[24] H. Bay, A Ess, T. Tuytelaars, and L. Gool. "SURF: Speeded Up Robust Features", Computer Vision and Image Understanding (CVIU). 2008, 110 (3), 346-359.

[25] OpenCV. *Feature Matching*. 2016.
http://docs.opencv.org/trunk/dc/dc3/tutorial_py_matcher.html.

[26] M. Selecký, and T Meiser. Integration of Autonomous UAVs into Multi-agent Simulation. Acta Polytechnica.. 2012, 52 (5/2012), 93-99.

[27] S M LaValle. *Planning algorithms*. 2012.
http://planning.cs.uiuc.edu/node102.html.

[28] T. Cormen, Leiserson C., R. Rivest, and C. Stein. *Introduction to algorithms*. 2002.

[29] A. Abramov. *Convex hull: how to tell whether a point is inside or outside?* 2016.
https://salzis.wordpress.com/2014/05/01/convex-hull-how-to-tell-whether-a-point-is-inside-or-outside/.

[30] P. Matuschek, J. *Finding Points Within a Distance of a Latitude/Longitude Using Bounding Coordinates*. 2016.
http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates.